

**USENIX**

**WASHINGTON, D. C. CONFERENCE PROCEEDINGS**

**USENIX**

**CONFERENCE  
PROCEEDINGS**

Washington, D. C.  
January 22 -26, 1990

**WINTER  
1990**





# **USENIX Association**

## **Proceedings of the Winter 1990 USENIX Conference**

**January 22 — January 26, 1990  
Washington, D.C., USA**

For additional copies of these proceedings contact  
USENIX Association  
2560 Ninth St., Suite 215  
Berkeley, CA 94710 USA

The price is \$25.  
Outside the U.S. and Canada please add  
\$15 per copy for postage (via air printed matter)

Copyright 1990 by The USENIX Association  
All rights reserved.

This volume is published as a collective work.  
Rights to individual papers remain  
with the author or the author's employer.

UNIX is a registered trademark of AT&T.  
Other trademarks are noted in the text.

## TABLE OF CONTENTS

Preface .....	vii
Conference Committee .....	ix
Author Index .....	xi
Permuted Index .....	xii

### PLENARY SESSION

**Wednesday (9:00-10:30)**

**Chair: Daniel Klein**

**WELCOME AND INTRODUCTIONS:**

*Daniel Klein, Software Engineering Institute, CMU; Ellie Young, USENIX Association*

**KEYNOTE ADDRESS: NASA's Manned Spacecraft Computers**

*Jim Tomayko, Software Engineering Institute, CMU*

### VIRTUAL MEMORY

**Wednesday (11:00-12:30)**

**Chair: Chet Juszczak**

A Dynamic File System Inode Allocation and Reclaim Policy .....	1
<i>Ronald E. Barkley, T. Paul Lee, AT&amp;T Bell Laboratories</i>	
Insuring Improved VM Performance: Some No-Fault Policies .....	11
<i>Danny Chen, Ronald E. Barkley, T. Paul Lee, AT&amp;T Bell Laboratories</i>	
TAE Plus: Transportable Applications Environment Plus A User Interface Development Tool for Building Graphic Oriented Applications .....	363
<i>Martha Szczur, Karl R. Wolf, NASA/Goddard Space Flight Center</i>	

## ARCHITECTURE & DEBUGGERS

**Wednesday (2:00-3:30)**

**Chair: John Mashey**

Implementing a Mach Debugger for Multithreaded Applications .....	25
<i>Deborah Caswell, Hewlett-Packard Company; David Black, Carnegie Mellon University</i>	
pdb: A Network Oriented Symbolic Debugger .....	41
<i>Paul Maybee, Solbourne Computer, Inc.</i>	
Some Efficient Architecture Simulation Techniques .....	53
<i>Robert Bedichek, University of Washington</i>	

## APPLICATIONS

**Wednesday (4:00-5:30)**

**Chair: Susanne Smith**

Tickerplants on UNIX .....	65
<i>Robert Berkley, Skip Gilbrech, Timothy Hunt, Mark Luppi, Richard Plevin, Fusion Systems Group</i>	
GENESIS and XODUS, General Purpose Neural Network Simulation Tool .....	75
<i>Matt Wilson, John Uhley, Upinder Bhalla, David Bilitch, Mark Nelson, James Bower, California Institute of Technology</i>	
Keynote - A Language and Extensible Graphical Editor for Music .....	89
<i>Tim Thompson, AT&amp;T Bell Laboratories</i>	

## UTILITIES

**Thursday (9:00-10:30)**

**Chair: John Devitofranceschi**

Integrated Interactive Access to Heterogeneous Distributed Services .....	101
<i>Joel S. Emer, Digital Equipment Corp.; William E. Wehl, Massachusetts Institute of Technology</i>	
The UNIX System Math Library, a Status Report .....	117
<i>Joel D. Silverstein, Steven E. Sommars, Yio-Chian Tao, AT&amp;T Bell Laboratories</i>	
Tcl: An Embeddable Command Language .....	133
<i>John K. Ousterhout, University of California at Berkeley</i>	

## KERNEL INTERNALS

**Thursday (11:00-12:30)**

**Chair: Charlie Perkins**

An Event-Based Fair Share Scheduler .....	147
<i>Raymond B. Essick, Prisma, Inc.</i>	
Parallel STREAMS: a Multi-Processor Implementation .....	163
<i>Arun Garg, Sequent Computer Systems</i>	
Implementing Berkeley Sockets in System V Release 4 .....	177
<i>Ian Vessey, Glenn Skinner, Sun Microsystems</i>	

## NETWORKS

**Thursday (2:00-3:30)**

**Chair: Alix Vasilatos**

Two Network Management Tools (How Many Packets Would a Packet Router Route if a Packet Router Could Route Packets?) .....	195
<i>Allan Leinwand, Jeff Okamoto, Hewlett-Packard</i>	
Traffic Characterization of the NSFNET National Backbone .....	207
<i>Steven A. Heimlich, University of Maryland</i>	
Pseudo-Network Drivers and Virtual Networks .....	229
<i>S.M. Bellovin, AT&amp;T Bell Laboratories</i>	

## ETHICS IN THE COMPUTER INDUSTRY

**Thursday (4:00-5:30)**

**Moderator: Rob Kolstad**

A panel composed of a lawyer, a CEO, an ethicist, and others will discuss various questions about ethics and the computing industry.

## USER INTERFACE MANAGEMENT SYSTEMS

**Friday (9:00-10:30)**

**Chair: Dan Geer**

Serpent: A User Interface Management System .....	245
<i>Len Bass, Erik Hardy, Rick Kazman, Robert Seacord, Carnegie Mellon University; Brian Clapper, Naval Air Development Center</i>	
Parallel Object-Oriented UIMS with Macro and Micro Stubs .....	259
<i>Masami Hagiya, Kouji Ohtani, Kyoto University</i>	
MTX - A Shell that Permits Dynamic Rearrangement of Process Connections and Windows .....	275
<i>Stephen A. Uhler, Bell Communications Research</i>	

## FILE SYSTEMS

**Friday (11:00-12:30)**

**Chair: Kirk McKusick**

Using Unix as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers .....	285
<i>David Hitz, Guy Harris, James K. Lau, Allan M. Schwartz, Auspex Systems</i>	
A Highly-Parallelized Mach-Based Vnode Filesystem .....	297
<i>Alan Langerman, Joseph Boykin, Susan LoVerso, Shashi Mangalat, Encore Computer Corporation</i>	
Disk Scheduling Revisited .....	313
<i>Margo Seltzer, Peter Chen, John Ousterhout, University of California, Berkeley</i>	

## LANGUAGES AND SOFTWARE ENGINEERING

**Friday (2:00-4:00)**

**Chair: Dan Klein**

Postloading for Fun and Profit .....	325
<i>S.C. Johnson, Stardent Computer Corp.</i>	
Multiple Site Source Reconciliation .....	331
<i>Dodi Francisco, Lois C. Price, TRW Financial Systems</i>	
CVS II: Parallelizing Software Development .....	341
<i>Brian Berliner, Prisma, Inc.</i>	
Ada and Binary Unix Standards .....	353
<i>Mitchell Gart, Alslys Inc.</i>	



## PREFACE

In 1976, I was handed a spool of magnetic tape by my boss. The tape contained the Version 5 Unix system, and he pointed to our PDP-11/40e and said "Get this running - the documentation is on the tape." In those days, a "standard" machine had 64 Kbytes of memory and seven megabytes of fixed and removeable disk space. We supported 16 terminals with that, and could actually get meaningful work done with 8 users logged in at once. If you were lucky enough to have a PDP-11/70 (an unbelievable luxury which we never acquired), you could actually run with more than a quarter megabyte of memory and more disk space than Kernel knew how to handle.

Four years later, I attended my first USENIX conference in Austin, where perhaps a couple of hundred of us gathered in the University of Texas auditorium, argued over inode cacheing in the Version 7 Kernel and whether switches should be added to *cat*, and attended the first USENIX vendor exhibition - four folding tables in a back room, a few handouts, and the President of Able Computer Corporation sitting on a table explaining optimal card layout on the Unibus.

The old times were fun and interesting, and we like to reminisce about them, but both Unix and USENIX have changed a bit since then. Unix has spread beyond any of our wildest imagining, being the operating system of choice for hundreds of thousands of machines. As a user's association, USENIX has changed from a handful of rugged individualists working on an operating system that hardware manufacturers disdained, to a member organization of over 4000 people worldwide. And I have changed - from a long haired bearded hacker to a not-so-long albeit thinner haired, bearded "computer scientist".

Although the people, personas, and perceptions have changed, one thing has remained constant - the ceaseless innovation and creativity of the people using, exploiting, and expanding the flexibility of Unix. These proceedings try to capture some of the best of that work.

When I was asked to chair the Winter 1990 USENIX Technical Conference, I fairly jumped at the prospect. And in spite of the warnings and remonstrations of my predecessors, and of the frequent headaches and narrowly averted catastrophes, I have enjoyed my task immensely. Unlike the previous three conferences (which required full papers for consideration and peer review), this conference asked for extended abstracts - a 2000-3000 word summarization of a full paper. Not only did this increase the strength and breadth of the submissions, but it enabled all the program committee members to review each submission, and allowed us to fairly evaluate what papers could be assembled to make the best conference possible. For my program committee, I assembled what I felt were the best technical and creative minds in the community. The task of paring the 83 initial submissions down to the 31 presented here was a difficult one, and this conference would not be the success it is without their invaluable knowledge and assistance.

This conference is different from past technical conferences in that we are adding a number of new, fun, and timely items to the program and proceedings. So that you can better identify them during the conference, photographs of the authors and program committee are being included in the proceedings. Our keynote speaker, Jim Tomayko, offers an entertaining and informative look at the NASA manned spaceflight computer systems from the earliest Mercury missions to the present day Shuttle. With the one year anniversary of the infamous RTM Internet Worm recently passed, on Thursday a special panel session will discuss various questions about ethics in the computer industry. And since our review and selection process concluded in early September,

Work-in-Progress sessions on Thursday and on Friday will showcase more recent developments. Finally, we have also introduced new concurrent sessions to the conference, so that in addition to formal, published presentations, we will also have a set of free tutorials throughout the week where people can exchange ideas and information in a more informal atmosphere. Attendees will be free to migrate between all sessions, and if there is sufficient interest, these new sessions will continue as a regular event.

I would like to thank the USENIX Association and its board of directors for providing us with this conference and for giving me the opportunity to be a part of it, and I would like to thank you all for attending and participating. Gracious and heartfelt thanks are also due to Judy DesHarnais for planning assistance and all for the thankless scut work that goes into making a conference like this work; to the Software Engineering Institute for allowing me the time and resources needed to be program chair; to Evi Nemeth and her tireless staff for producing these proceedings; to Ellie Young and Andrea Galleni for coordinating everything, offering good advice, and keeping me sane; and to Oscar and Azhyet, my two cats, for keeping my feet warm and purring at me when I needed it the most.

Please enjoy the conference – I am sure it will be a smashing success!

Daniel Klein  
Program Chair

## TECHNICAL PROGRAM COMMITTEE



**Eric P. Allman**  
*UC Berkeley*



**Pat Caruthers**  
*Aratar*



**John Devitofranceschi**  
*University of Illinois*



**Michelle Dominijanni**  
*Concurrent*



**Mike O'Dell**  
*Prisma*



**Dan Geer**  
*MIT - Project Athena*



**Chet Juszczak**  
*DEC*



**Daniel Klein, Chair**  
*CMU - SEI*



**John Mashey**  
*MIPS*



**Charlie Perkins**  
*IBM Watson*



**Dennis M. Ritchie**  
*AT&T Bell Labs*



**Susanne Smith**  
*Windsound*



**Alix Vasilatos**  
*OSF*

## CONFERENCE COMMITTEE

### CONFERENCE ORGANIZERS

Daniel Klein, *Technical Program Chair*  
(Sun Microsystems, Inc.)  
Judith F. Desharnais, *Meeting Planner*  
(USENIX Association)  
John Donnelly, *Tutorial Coordinator*  
(USENIX Association)  
Evi Nemeth, *Proceedings Production*  
(University of Colorado, Boulder)  
Sonya Neuffer, *Connectivity Specialist*  
(Canstar, Inc.)

### PROCEEDINGS PRODUCTION (University of Colorado)

Evi Nemeth  
Dotty Foerst  
Barb Dyker  
Darren Hardy  
Trent Hein  
Paul Kooros  
Laszlo Nemeth

### TECHNICAL PROGRAM REVIEWERS

Roberta Anslow  
Judy Bamberger  
Charlie Briggs  
Roman J. Budzianowski  
Heather Burris  
Chuck Clanton  
Mitch Condylis  
Tom Coppeto  
Susan Dart  
Al Delorey  
Greg Depp  
Jeff Dike  
Bob Dillberger  
John Dustin  
Peter Feiler  
Gary Gaudet  
Henry Hall  
Paul Holbrook  
Carolyn Lathrop  
Marshall Kirk McKusick  
Henry Mensch  
Patrick Place  
Jon Reeves  
Win Treese  
Jim Woodward

The USENIX Association would like to thank NewGen Systems Corporation and their Boulder, CO distributor Oblio, Inc. for use of the newgen TurboPS 400 printer used in producing part of these proceedings.

## AUTHOR INDEX

Ronald E. Barkley	1	James K. Lau	285
Ronald E. Barkley	11	T. Paul Lee	1
Len Bass	245	T. Paul Lee	11
Robert Bedichek	53	Allan Leinwand	195
Steven M. Bellovin	229	Susan LoVerso	297
Robert Berkley	65	Mark Luppi	65
Brian Berliner	341	Shashi Mangalat	297
Upinder Bhalla	75	Paul Maybee	41
David Bilitch	75	Mark Nelson	75
David Black	25	Kouji Ohtani	259
James Bower	75	Jeff Okamoto	195
Joseph Boykin	297	John K. Ousterhout	133
Deborah Caswell	25	John K. Ousterhout	313
Danny Chen	11	Richard Plevin	65
Peter Chen	313	Lois C. Price	331
Brian Clapper	245	Allan M. Schwartz	285
Joel S. Emer	101	Robert Seacord	245
Raymond B. Essick	147	Margo Seltzer	313
Dodi Francisco	331	Joel D. Silverstein	117
Arun Garg	163	Glenn Skinner	177
Mitchell Gart	353	Steven E. Sommars	117
Skip Gilbrech	65	Martha Szczur	363
Masami Hagiya	259	Yio-Chian Tao	117
Erik Hardy	245	Tim Thompson	89
Guy Harris	285	Stephen A. Uhler	275
Steven A. Heimlich	207	John Uhley	75
David Hitz	285	Ian Vessey	177
Timothy Hunt	65	William E. Weihl	101
Stephen C. Johnson	325	Matt Wilson	75
Rick Kazman	245	Karl R. Wolf	363
Alan Langerman	297		

## PERMUTED INDEX

Berkeley Sockets in System V Release	4	177
Integrated Interactive	Access to Heterogeneous Distributed Services	101
A Dynamic File System Inode	Ada and Binary Unix Standards	353
Development Tool for Building Graphic Oriented	Allocation and Reclaim Policy	1
Implementing a Mach Debugger for Multithreaded	Applications	363
TAE Plus: Transportable	Applications	25
Some Efficient	Applications Environment Plus A User Interface	363
Packet Trains on NSFNET National	Architecture Simulation Techniques	53
Implementing	Backbone A Traffic Characterization	207
Ada and	Berkeley Sockets in System V Release 4	177
Plus A User Interface Development Tool for	Binary Unix Standards	353
Trains on NSFNET National Backbone A Traffic	Building Graphic Oriented Applications	363
Tcl: An Embeddable	Characterization	207
Using Unix as One	Command Language	133
that permits dynamic rearrangement of process	Component of a Lightweight Distributed Kernel	285
pdb: A Network Oriented Symbolic	connections and Windows	275
Implementing a Mach	CVS-II: Parallelizing Software Development	341
CVS-II: Parallelizing Software	Debugger	41
Applications Environment Plus A User Interface	Debugger for Multithreaded Applications	25
Using Unix as One Component of a Lightweight	Development	341
Integrated Interactive Access to Heterogeneous	Development Tool for Building Graphic Oriented	363
Pseudo-Network	Disk Scheduling Revisited	313
A	Distributed Kernel for Multiprocessor File	285
MTX: A Shell that permits	Distributed Services	101
Keynote: A Language and Extensible Graphical	Drivers and Virtual Networks	229
Some	Dynamic File System Inode Allocation and	1
Tcl: An	dynamic rearrangement of process connections	275
TAE Plus: Transportable Applications	Editor for Music	89
An	Efficient Architecture Simulation Techniques	53
Keynote: A Language and	Embeddable Command Language	133
An Event-based	Environment Plus A User Interface Development	363
Distributed Kernel for Multiprocessor	Event-based Fair Share Scheduler	147
A Dynamic	Extensible Graphical Editor for Music	89
	Fair Share Scheduler	147
	File Servers	285
	File System Inode Allocation and Reclaim Policy	1



A Highly-Parallelized Mach-based Vnode	Filesystem	297
Postloading for	Fun and Profit	325
GENESIS and XODUS:	General Purpose Neural Network Simulation Tool	75
	GENESIS and XODUS: General Purpose Neural	75
A User Interface Development Tool for Building	Graphic Oriented Applications	363
Keynote: A Language and Extensible	Graphical Editor for Music	89
Integrated Interactive Access to	Heterogeneous Distributed Services	101
A	Highly-Parallelized Mach-based Vnode Filesystem	297
Two Network Management Tools	(How Many Packets Would a Packet Router Route	195
Parallel STREAMS: A Multi-Processor	Implementation	163
	Implementing a Mach Debugger for Multithreaded	25
	Implementing Berkeley Sockets in System V	177
Insuring	Improved VM Performance: Some No-Fault Policies	11
A Dynamic File System	Inode Allocation and Reclaim Policy	1
	Insuring Improved VM Performance: Some No-Fault	11
Integrated	Integrated Interactive Access to Heterogeneous	101
Applications Environment Plus A User	Interactive Access to Heterogeneous Distributed	101
The Serpent User	Interface Development Tool for Building Graphic	363
as One Component of a Lightweight Distributed	Interface Management System	245
	Kernel for Multiprocessor File Servers	285
Tcl: An Embeddable Command	Keynote: A Language and Extensible Graphical	89
Keynote: A	Language	133
The UNIX System Math	Language and Extensible Graphical Editor for	89
Using Unix as One Component of a	Library, a Status Report	117
Implementing a	Lightweight Distributed Kernel for	285
A Highly-Parallelized	Mach Debugger for Multithreaded Applications	25
Parallel Object-Oriented UIMS with	Mach-based Vnode Filesystem	297
The Serpent User Interface	Macro and Micro Stubs	259
Two Network	Management System	245
The UNIX System	Management Tools (How Many Packets Would a	195
Parallel Object-Oriented UIMS with Macro and	Math Library, a Status Report	117
	Micro Stubs	259
	MTX: A Shell that permits dynamic rearrangement	275
of a Lightweight Distributed Kernel for	Multiple Site Source Reconciliation	331
Parallel STREAMS: A	Multiprocessor File Servers	285
Implementing a Mach Debugger for	Multi-Processor Implementation	163
A Language and Extensible Graphical Editor for	Multithreaded Applications	25
Packet Trains on NSFNET	Music	89
Two	National Backbone A Traffic Characterization	207
pdb: A	Network Management Tools (How Many Packets	195
GENESIS and XODUS: General Purpose Neural	Network Oriented Symbolic Debugger	41
Pseudo-Network Drivers and Virtual	Network Simulation Tool	75
GENESIS and XODUS: General Purpose	Networks	229
Insuring Improved VM Performance: Some	Neural Network Simulation Tool	75
	No-Fault Policies	11

Packet Trains on	NSFNET National Backbone A Traffic	207
Parallel	Object-Oriented UIMS with Macro and Micro Stubs	259
Interface Development Tool for Building Graphic	Oriented Applications	363
pdb: A Network	Oriented Symbolic Debugger	41
Many Packets Would a Packet Router Route if a	Packet Router Could Route Packets?)	195
Management Tools (How Many Packets Would a	Packet Router Route if a Packet Router Could	195
Router Route if a Packet Router Could Route	Packet Trains on NSFNET National Backbone A	207
Two Network Management Tools (How Many	Packets?)	195
	Packets Would a Packet Router Route if a Packet	195
	Parallel Object-Oriented UIMS with Macro and	259
	Parallel STREAMS: A Multi-Processor	163
CVS-II:	Parallelizing Software Development	341
	pdb: A Network Oriented Symbolic Debugger	41
Insuring Improved VM	Performance: Some No-Fault Policies	11
MTX: A Shell that	permits dynamic rearrangement of process	275
Plus: Transportable Applications Environment	Plus A User Interface Development Tool for	363
TAE	Plus: Transportable Applications Environment	363
Insuring Improved VM Performance: Some No-Fault	Policies	11
File System Inode Allocation and Reclaim	Policy	1
	Postloading for Fun and Profit	325
A Shell that permits dynamic rearrangement of	process connections and Windows	275
Postloading for Fun and	Profit	325
	Pseudo-Network Drivers and Virtual Networks	229
GENESIS and XODUS: General	Purpose Neural Network Simulation Tool	75
MTX: A Shell that permits dynamic	rearrangement of process connections and	275
A Dynamic File System Inode Allocation and	Reclaim Policy	1
Multiple Site Source	Reconciliation	331
Implementing Berkeley Sockets in System V	Release 4	177
The UNIX System Math Library, a Status	Report	117
Disk Scheduling	Revisited	313
Tools (How Many Packets Would a Packet Router	Route if a Packet Router Could Route Packets?)	195
a Packet Router Route if a Packet Router Could	Route Packets?)	195
Packets Would a Packet Router Route if a Packet	Router Could Route Packets?)	195
Tools (How Many Packets Would a Packet	Router Route if a Packet Router Could Route	195
An Event-based Fair Share	Scheduler	147
Disk	Scheduling Revisited	313
The	Serpent User Interface Management System	245
Distributed Kernel for Multiprocessor File	Servers	285
Interactive Access to Heterogeneous Distributed	Services	101
An Event-based Fair	Share Scheduler	147
MTX: A	Shell that permits dynamic rearrangement of	275
Some Efficient Architecture	Simulation Techniques	53
and XODUS: General Purpose Neural Network	Simulation Tool	75
Multiple	Site Source Reconciliation	331
Implementing Berkeley	Sockets in System V Release 4	177

CVS-II: Parallelizing	Software Development	341
Multiple Site	Software Tickerplants on UNIX	65
Ada and Binary Unix	Source Reconciliation	331
The UNIX System Math Library, a	Standards	353
Parallel	Status Report	117
Object-Oriented UIMS with Macro and Micro	STREAMS: A Multi-Processor Implementation	163
pdb: A Network Oriented	Stubs	259
The Serpent User Interface Management	Symbolic Debugger	41
A Dynamic File	System	245
The UNIX	System Inode Allocation and Reclaim Policy	1
Implementing Berkeley Sockets in	System Math Library, a Status Report	117
	System V Release 4	177
	TAE Plus: Transportable Applications	363
Some Efficient Architecture Simulation	Tcl: An Embeddable Command Language	133
Software	Techniques	53
General Purpose Neural Network Simulation	Tickerplants on UNIX	65
Environment Plus A User Interface Development	Tool	75
Two Network Management	Tool for Building Graphic Oriented Applications	363
Packet Trains on NSFNET National Backbone A	Tools (How Many Packets Would a Packet Router	195
Packet	Traffic Characterization	207
User Interface Development Tool for	Trains on NSFNET National Backbone A Traffic	207
Parallel Object-Oriented	TAE Plus:	363
Software Tickerplants on	UIMS with Macro and Micro Stubs	259
Using	UNIX	65
Ada and Binary	Unix as One Component of a Lightweight	285
The	Unix Standards	353
Transportable Applications Environment Plus A	UNIX System Math Library, a Status Report	117
The Serpent	User Interface Development Tool for Building	363
	User Interface Management System	245
Implementing Berkeley Sockets in System	Using Unix as One Component of a Lightweight	285
Pseudo-Network Drivers and	V Release 4	177
Insuring Improved	Virtual Networks	229
A Highly-Parallelized Mach-based	VM Performance: Some No-Fault Policies	11
rearrangement of process connections and	Vnode Filesystem	297
GENESIS and	Windows	275
	XODUS: General Purpose Neural Network	75

1	THE JAPANESE LANGUAGE
2	THE JAPANESE LANGUAGE
3	THE JAPANESE LANGUAGE
4	THE JAPANESE LANGUAGE
5	THE JAPANESE LANGUAGE
6	THE JAPANESE LANGUAGE
7	THE JAPANESE LANGUAGE
8	THE JAPANESE LANGUAGE
9	THE JAPANESE LANGUAGE
10	THE JAPANESE LANGUAGE
11	THE JAPANESE LANGUAGE
12	THE JAPANESE LANGUAGE
13	THE JAPANESE LANGUAGE
14	THE JAPANESE LANGUAGE
15	THE JAPANESE LANGUAGE
16	THE JAPANESE LANGUAGE
17	THE JAPANESE LANGUAGE
18	THE JAPANESE LANGUAGE
19	THE JAPANESE LANGUAGE
20	THE JAPANESE LANGUAGE
21	THE JAPANESE LANGUAGE
22	THE JAPANESE LANGUAGE
23	THE JAPANESE LANGUAGE
24	THE JAPANESE LANGUAGE
25	THE JAPANESE LANGUAGE
26	THE JAPANESE LANGUAGE
27	THE JAPANESE LANGUAGE
28	THE JAPANESE LANGUAGE
29	THE JAPANESE LANGUAGE
30	THE JAPANESE LANGUAGE
31	THE JAPANESE LANGUAGE
32	THE JAPANESE LANGUAGE
33	THE JAPANESE LANGUAGE
34	THE JAPANESE LANGUAGE
35	THE JAPANESE LANGUAGE
36	THE JAPANESE LANGUAGE
37	THE JAPANESE LANGUAGE
38	THE JAPANESE LANGUAGE
39	THE JAPANESE LANGUAGE
40	THE JAPANESE LANGUAGE
41	THE JAPANESE LANGUAGE
42	THE JAPANESE LANGUAGE
43	THE JAPANESE LANGUAGE
44	THE JAPANESE LANGUAGE
45	THE JAPANESE LANGUAGE
46	THE JAPANESE LANGUAGE
47	THE JAPANESE LANGUAGE
48	THE JAPANESE LANGUAGE
49	THE JAPANESE LANGUAGE
50	THE JAPANESE LANGUAGE
51	THE JAPANESE LANGUAGE
52	THE JAPANESE LANGUAGE
53	THE JAPANESE LANGUAGE
54	THE JAPANESE LANGUAGE
55	THE JAPANESE LANGUAGE
56	THE JAPANESE LANGUAGE
57	THE JAPANESE LANGUAGE
58	THE JAPANESE LANGUAGE
59	THE JAPANESE LANGUAGE
60	THE JAPANESE LANGUAGE
61	THE JAPANESE LANGUAGE
62	THE JAPANESE LANGUAGE
63	THE JAPANESE LANGUAGE
64	THE JAPANESE LANGUAGE
65	THE JAPANESE LANGUAGE
66	THE JAPANESE LANGUAGE
67	THE JAPANESE LANGUAGE
68	THE JAPANESE LANGUAGE
69	THE JAPANESE LANGUAGE
70	THE JAPANESE LANGUAGE
71	THE JAPANESE LANGUAGE
72	THE JAPANESE LANGUAGE
73	THE JAPANESE LANGUAGE
74	THE JAPANESE LANGUAGE
75	THE JAPANESE LANGUAGE
76	THE JAPANESE LANGUAGE
77	THE JAPANESE LANGUAGE
78	THE JAPANESE LANGUAGE
79	THE JAPANESE LANGUAGE
80	THE JAPANESE LANGUAGE
81	THE JAPANESE LANGUAGE
82	THE JAPANESE LANGUAGE
83	THE JAPANESE LANGUAGE
84	THE JAPANESE LANGUAGE
85	THE JAPANESE LANGUAGE
86	THE JAPANESE LANGUAGE
87	THE JAPANESE LANGUAGE
88	THE JAPANESE LANGUAGE
89	THE JAPANESE LANGUAGE
90	THE JAPANESE LANGUAGE
91	THE JAPANESE LANGUAGE
92	THE JAPANESE LANGUAGE
93	THE JAPANESE LANGUAGE
94	THE JAPANESE LANGUAGE
95	THE JAPANESE LANGUAGE
96	THE JAPANESE LANGUAGE
97	THE JAPANESE LANGUAGE
98	THE JAPANESE LANGUAGE
99	THE JAPANESE LANGUAGE
100	THE JAPANESE LANGUAGE

# A Dynamic File System Inode Allocation and Reclaim Policy

Ronald E. Barkley  
AT&T Bell Laboratories  
Summit, NJ 07901  
attunix!rebark

T. Paul Lee†  
AT&T Bell Laboratories  
Holmdel, NJ 07733

## ABSTRACT

In this paper we describe a policy for dynamically allocating and reclaiming inodes and present some experimental results obtained on a System V prototype. Traditionally, in-core inodes are allocated from a statically preconfigured table, and free slots in this table are reused based on a LRU re-allocation policy. With a dynamic policy, inodes are allocated on demand and cached after use. We limit the number of inodes by reclaiming them when they no longer have any memory pages associated with them. This policy improves the efficiency of the memory subsystem and, thus, improves overall system performance. Dynamically allocating inodes also offers the advantage of automatic tuning; we do not need any special understanding of the workload to determine the best number of inodes to configure.

## 1. MOTIVATION

The internal representation of a UNIX® System V file is given by an *inode*, which contains information about the file such as the owner, the access permissions, the access times, and a description of the layout of the file on the disk<sup>[1]</sup>. Inodes exist in static form on the disk; the kernel reads the disk inodes into an in-core inode structure to use them. Because inodes are frequently reused and because disk accesses are expensive, inodes are normally cached in memory in an inode table. This table of inode structures is configured at system boot time, and slots in the table are allocated using a least recently used (LRU)<sup>[2]</sup> replacement algorithm. Valid entries in the inode table are kept on hash lists to make lookup operations efficient. Recently, the notion of an inode has been generalized to a *vnode* (virtual node) to accommodate multiple file system types and implementations<sup>[3]</sup>. In the System V *s5* file system type, the *vnode* structure is embedded in the inode structure and, thus, in this paper we use the terms "inode" and "vnode" somewhat interchangeably.

In the work reported in Rodriguez, *et al.*<sup>[4]</sup> many kernel data structures including the in-core inodes are allocated dynamically. Because inodes are cached, the number of dynamically allocated in-core inodes can grow indefinitely unless some policy is imposed to remove

---

† Author's current address: Sony Microsystems Company, 651 River Oaks Parkway, San Jose, California 95134



unnneeded but cached inodes. In that work the policy is simply to impose a hard upper limit on the number of dynamic inodes. If creating a new inode would exceed this upper limit, a new inode structure is not dynamically allocated; instead, the least recently used inode structure already allocated is reused.

We observed in our System V Release 4.0 performance measurements that system performance is sensitive to the number of inodes statically configured at boot time, and we traced the cause back to the VM subsystem<sup>[5]</sup>. In VM, physical pages with valid file system data are identified by their *<vnode, offset>* pair, where *vnode* is the address of the file system independent in-core structure representing the file. Pages with valid data that are not currently being used retain their identity and are placed on a page cache list. When a file system needs a new inode slot, it reuses a cached inode, and all pages associated with that inode (*vnode*) must lose their identity. Future references to these pages will require a disk read even though a valid copy might exist in the page pool, albeit without an identity. To make matters worse, the inode slot reuse is in LRU order without any consideration of the number of pages associated with the inode. The effectiveness of the page cache is thus artificially limited by the number of inode table entries configured at boot time.

## 2. VM PAGE CACHE MANAGEMENT

Before discussing the dynamic inode allocation and reclaim policy, we first need to describe how VM manages the system's physical memory pages. Each page in the system is either busy or free. Busy pages are those currently being used by the kernel or by user processes. Free pages are not currently being used, but they may still have valid contents that could be reused later. All pages that contain valid file system data, regardless of whether they are busy or free, are identified by a *<vnode, offset>* pair that represents the file and offset within the file for the data.

VM keeps two lists of free pages, the free list and the cache list. The free list is the list of pages that are completely free; these pages have no identity and their current contents have no value. For example, old stack or old dirty data pages are on the free list. The cache list is the list of pages that are not currently being used but whose contents represent valid pages from some file system. For example, text pages and clean data pages from processes that have died are on the cache list, as are file system data pages from files that were mapped into processes that have since exited. The cache list is maintained using a LRU policy.

Whenever the system needs a page from some file system, that is, from a particular *vnode* at a particular offset, it first checks with VM to see if that page already exists in memory. If a page with the correct identity is found, it is used. If it cannot be found, the file system reads the data from the physical device into a fresh, empty page frame. Empty pages are obtained using the VM function *pageget()*. *Pageget()* initially takes pages from the free list since these have no current value. When the free list is exhausted, *pageget()* takes pages from the cache list, starting with the oldest ones. Thus, pages associated with a file will eventually be stolen for a new use if they are not reused in time.

The *vnode* that is used to identify pages is actually only the memory address of the *vnode* structure. If the *vnode* structure is reused for a different *vnode*, all pages that are associated with the original *vnode* must be flushed to prevent them from accidentally being associated with the new *vnode*. VM purges all pages' references to the *vnode*, writes the pages back to the file system if they have been modified, takes them off the cache list, and places them on the free list. Any future reference to a page from the original *vnode* and offset will have to be



satisfied by reading the physical device, even if the page on the free list has not yet been reused, because the page can no longer be identified.

In systems with statically preconfigured vnodes, pages tend to outlive their associated vnodes because of some complex subsystem interactions. In particular, System V Release 4.0 provides a directory name lookup cache (DNLC) to speed the lookup operations for commonly used file pathnames. File names and their associated vnodes are held by the cache, and cache entries are recycled using a LRU policy. File systems may choose whether to use the DNLC. For file systems that do use it, their vnodes that are in the cache are kept active. Since the vnodes in the cache tend to be directories, they generally have few pages associated with them. The DNLC boosts lookup performance, but for file systems with statically preallocated vnodes, it also reduces the pool of vnodes in the file system's vnode free list. Thus, the vnode structures in the free list are recycled more quickly; unfortunately, these tend to be the vnodes for files that have many pages, such as executable and data files. Each time one of these vnode structures is reused, all the pages associated with the original vnode must be flushed, and they all lose their identities.

### 3. A DYNAMIC ALLOCATION AND RECLAIM POLICY

To avoid limiting the page cache effectiveness because the inode table is too small, we can over-configure the system, preconfiguring more inodes than we will ever need. However, this strategy wastes memory, and a better approach is to allocate in-core inode slots dynamically. In this way we allocate only what we need, but we can also keep them as long as necessary.

Dynamically allocating and reclaiming inode structures is a file system specific issue. This discussion of the policy and kernel changes to support dynamic inodes is based on the System V *s5* file system type. However, the policy can easily be adopted to other types.

#### 3.1 Allocating and Reclaiming an Inode Structure

The dynamic policy contains two parts: an allocation policy and a reclaim policy. First, we no longer allocate any inode structures at system initialization time. Instead, we allocate memory for each inode structure on demand by requesting it from the kernel memory allocator, and we cache these in-core inode structures to avoid having to re-read the inode from the disk.

To prevent the number of inode structures (both active and cached) from growing unchecked we need a reclaiming policy. The reclaim policy is tied to the number of memory pages that are associated with each inode structure. The VM page cache list manager and the page aging and stealing daemon (*pageout*) are responsible for reclaiming and stealing pages so that they can be used where they are most needed. If an inode is not being used and is unneeded, the pages associated with it will gradually be taken away. When all its pages are gone, the inode is considered *inactive* and the inode reclaim policy will flush the inode from the system. The memory for cached inode structures without any pages on their page lists can either be freed back to the kernel memory allocator or be reused for a new inode.

#### 3.2 Kernel Changes

This section describes the changes to *s5* needed to support the dynamic inode allocation and reclaim policy. The major changes are to the *iget()* function that allocates an inode and to the *ipfree()* function that releases the inode.

*Iget()* is called when the system needs the inode for a particular file. In *iget()*, we first look in the inode cache (using the same hashing algorithm that the static system uses) to see if the

inode is already present. If it is not, we need a new inode structure. We take the *first* inode from the inode free list and see if there are any pages still associated with the inode. If there are no pages, we reuse the inode. If there are pages still associated with the inode, we put the inode back on the *end* of the free list and then call the kernel memory allocator to allocate a new inode structure. Although we could generalize this scheme to examine the first  $N$  inodes in the free list, we use the simple heuristic of only looking at the first inode because it is fast and we found it to be efficient in purging unneeded inodes from the cache.

If an inode still has pages associated with it when it is being freed in *ipfree()*, we link it to the end of the free list. (We also put the inode on the free list if it is needed by some other process, if it is a swap file, or if it has been recursively locked.) Otherwise, if the inode does not have any pages, we free the entry and return the memory back to the kernel memory allocator.

In addition to these changes, we also change *s5update()* and *iflush()* where previously we traversed the inode table as an array. This code must be changed because with dynamic inodes we no longer have physically contiguous inode slots. Instead of walking through the static array of inode entries, we now walk through the list in each inode hash bucket. The *s5unmount()* function must also be modified to free and unhash the root inode of the unmounted file system.

Finally, the inode table overflow flag that was used by the system activity reporter (*sar(1)*) becomes obsolete with this new policy.

#### 4. EXPERIMENTS AND ANALYSIS OF RESULTS

We implemented a prototype of the dynamic inode allocation and reclaim policy on an AT&T microcomputer to gather empirical performance data and compare it with the original system with a static (preconfigured) inode table. To measure the system, we used a multi-user timesharing workload benchmark<sup>[6]</sup>.

All measurements were made on a system with 4MB memory, running a development load of System V Release 4.0. During all experiments, the size of the DNLC was held at 300 entries.

##### 4.1 System Times

We collected data for the baseline system with 200, 400, and 600 entries in the static inode table and then collected the same data for the system with dynamically allocated inodes. The system times for the different systems (as a function of the number of simulated user scripts) are shown in Figure 1. The system times reflect the CPU time spent in kernel mode managing shared resources and are clearly sensitive to the number of inodes configured. With 12 user scripts, the savings relative to 200 inodes are 12% for 600 inodes and 16% for dynamically allocated inodes, respectively. For this workload, configuring more than 600 inode entries for the static system will not significantly improve the system times. In addition to costing less than even 600 static inodes, using dynamically allocated inodes offers the advantage of "automatic tuning," i.e., we do not need any special understanding of the workload to determine the best number of inodes to configure.

##### 4.2 Dynamic Behavior of Inode Usage

We also sampled the total number of inodes and the number of free inodes in the dynamic system at 10-second intervals during the run of 10 user scripts. These results are shown in Figure 2.

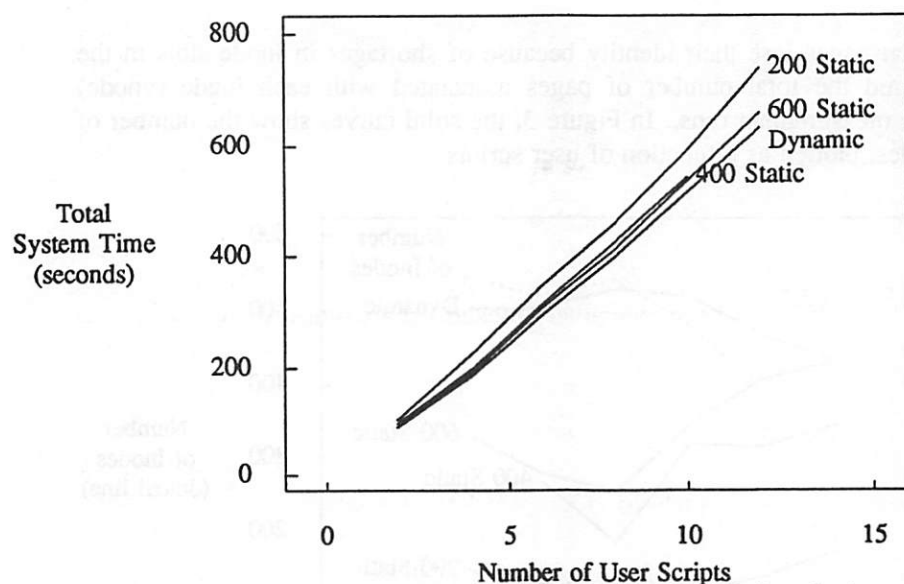


Figure 1. System Times As a Function of User Scripts

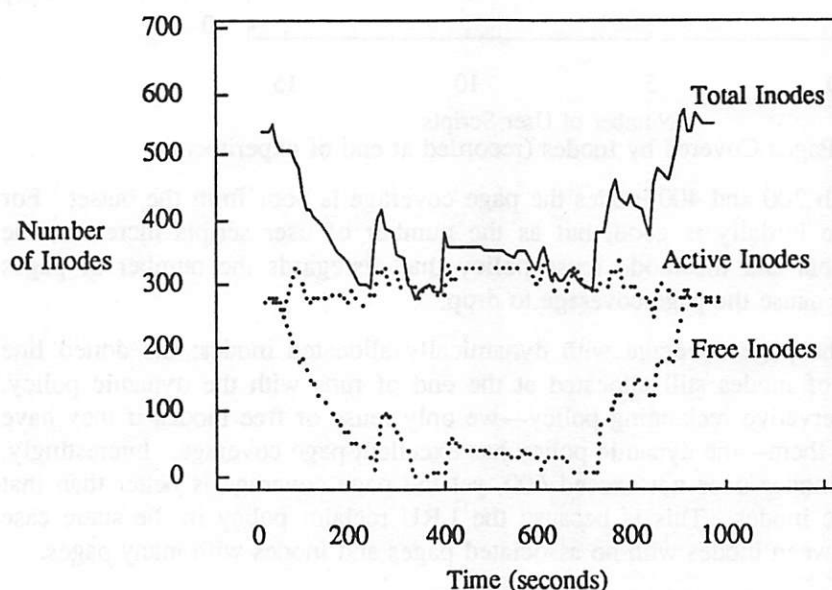


Figure 2. Inode Usage Time Series

By definition, the total number of inodes equals the sum of the number of active inodes and the number of free inodes. The trace shows that the number of active inodes does not change much during the experiment although the members in the active set change as evidenced by the reuse of the free inodes. The free inodes are being reclaimed efficiently since the total number of inodes does not increase much at the height of the measurement run; as expected, the page stealing and reclaiming policy frees pages and, thus, indirectly frees inodes that are no longer needed. At the end of the benchmark run, pages with valid file system data are not being reclaimed, and thus, they remain in the page cache. Consequently, the number of free inodes increases because the pages in the cache list are still associated with them.

### 4.3 Page Coverage

To test our hypothesis that pages lose their identity because of shortages in inode slots in the static system, we measured the total number of pages associated with each inode (vnode) structure at the end of the measurement runs. In Figure 3, the solid curves show the number of pages covered by all inodes, plotted as a function of user scripts.

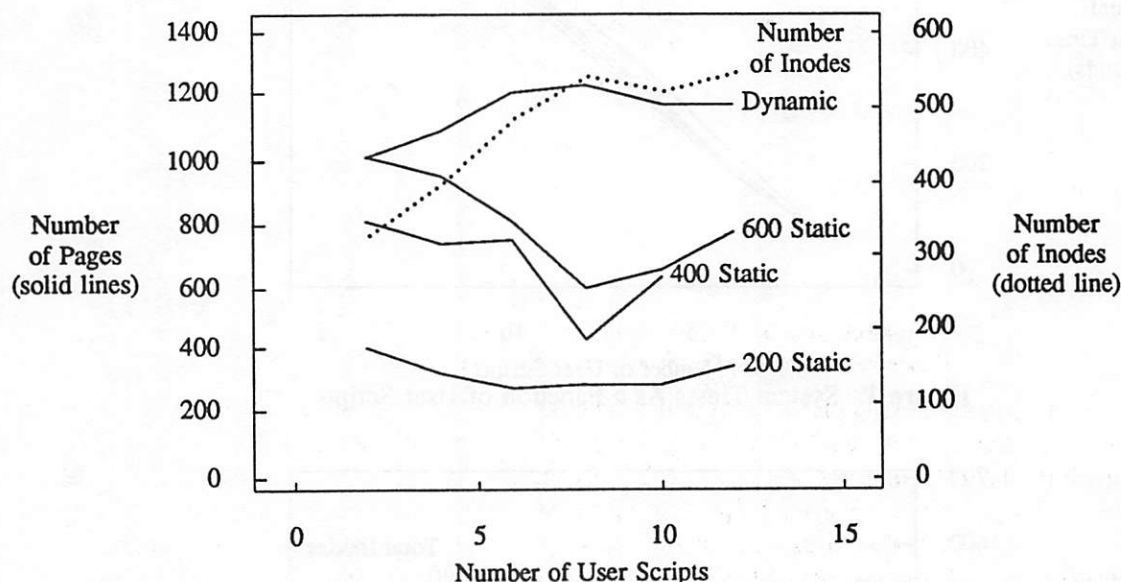


Figure 3. Pages Covered by Inodes (recorded at end of experiment)

In the static case, for both 200 and 400 inodes the page coverage is poor from the outset. For 600 inodes, the coverage initially is good, but as the number of user scripts increases, the competition for inode slots and the inode reuse policy that disregards the number of pages associated with the inode cause the page coverage to drop.

The graph also shows the page coverage with dynamically allocated inodes; the dotted line shows the total number of inodes still allocated at the end of runs with the dynamic policy. Because we have a conservative reclaiming policy—we only reuse or free inodes if they have no pages associated with them—the dynamic policy has excellent page coverage. Interestingly, the number of dynamic inodes does not exceed 600, yet the page coverage is better than that obtained using 600 static inodes. This is because the LRU reclaim policy in the static case does not discriminate between inodes with no associated pages and inodes with many pages.

### 4.4 Clean and Modified Page Distribution

It is also interesting to analyze the number and type of pages covered by the inodes. The *type* of a page is either *clean* or *modified*; modified pages have to be written back to the disk when the inode is reused or the page is reclaimed. Table 1 shows the frequency of occurrence as a function of the total number of pages and the number of modified pages for a particular inode. The value of the  $(m, t)$  entry in the table is the number of inodes that have  $m$  modified pages and  $t$  total pages. Since the total number of pages includes the number of modified pages, the table has a triangular shape. The distribution is skewed; many files have a small number of pages in-core and most of them are clean (not modified) pages. This distribution helps justify the simple heuristic of only looking at the first inode when examining the free list; it is likely that the first free inode will have no pages associated with it and, thus, will be available for



TABLE 1. Number of Inodes Having  $t$  Pages and  $m$  Modified Pages

Modified ( $m$ )	Total ( $t$ )												
	0	1	2	3	4	6	8	12	16	24	32	48	64
0	11935	13868	2179	1427	1571	311	945	177	600	61	45	21	40
1			561	58	4	6							
2				264	191	3							
3					71	42	3						
4						22							
6									2	3			
8								2		5			
12									2	12	3		
16										21	15	16	7
24										55	99	98	23
32											3	14	12
48													5
64													

Numbers are sums of observed values at 10-seconds intervals.  
Data obtained during a benchmark run with 10 user scripts.

reuse. It is particularly interesting to note that inodes tend to have either all clean pages or mostly modified pages. This reflects the usage of files by the workload, which either opens files for reading and execution or opens them for writing. The data is shown graphically in Figure 4 with the probability density heights exaggerated using a logarithmic scale. The two ridges with the valley between illustrates the all-clean or mostly-modified observation above. More persistent and frequently modified files such as databases might exhibit a different distribution.

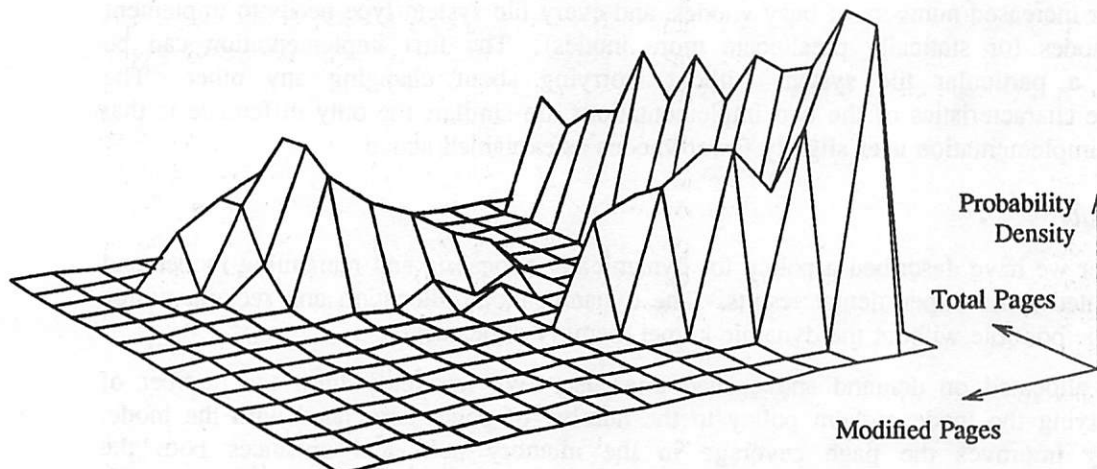


Figure 4. Probability Density for Total and Modified Pages (log scale on all axes)

## 5. AN ALTERNATE IMPLEMENTATION

In the implementation discussed in Section 3, the file system dependent code is responsible for detecting when vnodes still have pages associated with them and deciding when to free or reuse a vnode structure. Vnodes that still have pages but are not being used by any process are kept on a free list, but the scheme will only use a vnode structure from that free list for a new vnode when all the pages associated with the original vnode have disappeared.

In this section we discuss an alternate implementation for holding the vnodes while there are still valid pages associated with them. Each vnode has a reference count that counts the current "users" of the vnode. Whenever the count drops to 0, the vnode is inactivated. The file system may put the vnode structure on a free list (where it may be reused for another vnode), or it may free the memory allocated for the structure. Traditionally, a "user" is a process accessing the vnode. If, however, we also allow each page associated with the vnode to be considered a "user," the vnode will remain active as long as it has any pages, and we guarantee that no vnode will be reused until all its pages have disappeared.

The VM subsystem has two routines, *page\_hashin()* and *page\_hashout()*, that are used to add a page to and remove a page from a vnode's page list. To implement this new strategy, we add a call to *VN\_HOLD()* in *page\_hashin()* to increment the reference count when we add a page. We also add a call to *VN\_RELE()* in *page\_hashout()* to decrement the reference count (and possibly inactivate the vnode if the reference count is now 0) when we remove a page. The changes to *iget()* and *ipfree()* to support dynamic inode allocation become simpler. In *ipfree()*, we know that vnodes are never freed until they have no pages, and thus, except for the unusual conditions noted earlier, we always free the memory back to the kernel memory allocator. When *iget()* is called to get a new vnode, we still check the free list. It is almost always empty, but if there is a vnode in the list, it will not have any pages and we can reuse its vnode structure. Normally, however, we have to allocate memory for a fresh structure.

This implementation is simpler than the one discussed in Section 3. It also has the advantage that vnodes are released as soon as the last page disappears. In the first implementation, vnodes in the middle of the free list whose pages have all been stolen continue to exist until they bubble to the head of the free list and are reused for a new vnode; this results in some unnecessary additional vnodes. However, the second implementation includes changes to the VM subsystem, not just to the file system specific code. Thus, all the file systems must be prepared for increased numbers of busy vnodes, and every file system type needs to implement dynamic inodes (or statically preallocate more inodes). The first implementation can be applied to a particular file system without worrying about changing any other. The performance characteristics of the two implementations are similar; the only difference is that the second implementation uses slightly fewer vnodes as explained above.

## 6. SUMMARY

In this paper we have described a policy for dynamically allocating and reclaiming inodes and have presented some experimental results. The dynamic inode allocation and reclaim policy would not be possible without the dynamic kernel memory allocator<sup>[7]</sup>.

Inodes are allocated on demand and cached after use. We implicitly limit the number of inodes by tying the inode reclaim policy to the number of pages associated with the inode. This policy improves the page coverage in the memory pool and enhances both the performance of the memory subsystem and of the overall system. We do not need to search or sort the free inodes according to the number of pages linked to them. Instead, we use a simple heuristic of recycling inodes in the free list one at a time if they have any pages associated with them. This saves time without leaving too many unneeded inodes in memory. A large static allocation of inodes will not work unless it is coupled with an inode reclaim policy better than a simple LRU scheme. For example, it could search the free list to reclaim the inode slot with the fewest pages attached to it, but this strategy would be expensive. The dynamic inode allocation and reclaim policy is file system specific, and our prototype was based on the *s5* file system type. However, the policy can be easily adapted to other types.



We also discuss an alternate implementation based on the vnode's reference count. This implementation is simpler, but it requires changes to the file system independent part of the kernel; thus, all file system types have to either statically preconfigure more inode slots or incorporate a dynamic allocation and reclaim policy.

## 7. ACKNOWLEDGEMENTS

We would like to thank Dave Presotto for his helpful comments on this paper and for suggesting the alternate implementation described in Section 5.

## REFERENCES

1. M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.
2. R. L. Mattson, J. Gessei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, Vol. 9, No. 2, 1970, pp. 79-117.
3. S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in SUN UNIX," *Proc. of the 1986 Summer USENIX Conf.*, June 1988, pp. 238-247.
4. R. Rodriguez, M. Koehler, L. Palmer and R. Palmer, "A Dynamic UNIX Operating System," *Proc. of the 1988 Summer USENIX Conf.*, June 1988, pp. 305-319.
5. R. A. Gingell, J. P. Moran, and W. A. Shannon, "Virtual Memory Architecture in SunOS," *Proc. of the 1987 Summer USENIX Conf.*, June 1987, pp. 81-94.
6. S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities," *Conf. Proc. of CMG XIII*, December 1982, pp. 62-67.
7. T. P. Lee and R. E. Barkley, "A Watermark-based Lazy Buddy System for Kernel Memory Allocation," *Proc. of the 1989 Summer USENIX Conf.*, June 1989, pp. 1-13.



Ronald Barkley  
AT&T

Ronald Barkley has been with AT&T Bell Laboratories since 1984. As a member of the UNIX System Development Department, Ron was involved in the performance analysis and improvement of System V Releases 3 and 4, especially in the area of memory management. He is currently working on the analysis of AT&T's implementation of the OSI protocol stack. Ron has a MS degree in mathematics from Clemson University and a MS degree in computer science from Rutgers.



T. Paul Lee  
AT&T

T. Paul Lee joined Bell Labs in 1984 after earning his Ph.D. degree in computer science from the University of California at Berkeley. At Bell Labs, Paul was active in the performance analysis of the UNIX operating system, most recently working on performance improvements for UNIX System V Release 4.0. He is currently employed at Sony Microsystems in San Jose, California. Before his graduate work at Berkeley, Paul worked for Motorola and Bell Northern Research in the area of microprocessor design and firmware programming.



## Insuring Improved VM Performance: Some No-Fault Policies

*Danny Chen*  
*AT&T Bell Laboratories*  
*Holmdel, New Jersey*  
*att!hocus!dwc*

*Ronald E. Barkley*  
*AT&T Bell Laboratories*  
*Summit, New Jersey*  
*attunix!rebark*

*T. Paul Lee†*  
*AT&T Bell Laboratories*  
*Holmdel, New Jersey*

### *Abstract*

The major features of AT&T's UNIX<sup>®</sup> System V and Sun Microsystems' SunOS<sup>™</sup> have been merged into UNIX System V Release 4.0. Since the two systems have different and incompatible memory management architectures (**regions** for System V and **VM** for SunOS), it was decided that System V adopt the **VM** architecture. This decision is attributable to **VM**'s greater flexibility in managing process address spaces. Unfortunately, the initial port of Sun's **VM** implementation onto AT&T's 3B2/400 system showed that with this flexibility came an increase in overhead relative to a similar **regions**-based system.

We have found that two major contributing factors in the increase in overhead are an increase in the cost of handling page faults and an increase in the number of page faults required to execute processes. In this paper, we describe changes we have made in the *implementation* of the **VM** subsystem (i.e. not the *architecture*) to reduce the number of faults needed to execute processes. We present data to show the performance improvements we have been able to obtain through these changes. Of particular significance is a technique we use to

---

† Author's current address: Sony Microsystems Company, 651 River Oaks Parkway, San Jose, California 95134

avoid copy on write faults after *forks*. We conclude with suggestions for future work. The changes described in this paper have been incorporated into System V Release 4.0.

## 1. Introduction

In early 1988, AT&T's Data System Group and Sun Microsystems agreed to merge the major features of UNIX<sup>®</sup> System V and SunOS<sup>™</sup> (a popular variant of the Berkeley implementations of the UNIX System). A decision had to be made with respect to the address space/memory management subsystems for the merged product. AT&T's UNIX System V Release 3 used the **regions** architecture to handle process address spaces while SunOS used the VM architecture. The two architectures are largely incompatible because of their different philosophies on the use of major data structures. After technical review of both architectures, it was decided that the VM architecture was more flexible in handling process address spaces and more successful in localizing hardware-specific memory management functions. VM was therefore adopted for Release 4.0.

One disadvantage of the **regions** architecture is the restriction on the alignment of a *region*, which is the basic unit of allocation of address space. Each *region* has to be aligned on a hardware segment boundary (page size  $\times$  number of entries in a page table). Under VM, the unit of allocation of address space is called a *segment* and the only restriction on the alignment of *segments* is that they fall on page boundaries. Another problem with **regions** is that the page table/disk block descriptor data structure for a page is restricted to handling pages that are either in memory or on the disk. Through the implementation of *segment* drivers, VM's architecture allows one to map almost any object into a process address space.

Unfortunately, measurements from the initial port of Sun's VM code onto the AT&T 3B2/400 showed that with this flexibility came a cost in performance. Measurements of the system running a multi-user benchmark showed that under the VM-based system processes incurred more page faults and that these page faults were more expensive than those under **regions**.

In the following sections, we describe and contrast the salient features of the **regions** and VM architectures and explain the reasons for the high overhead of VM on 3B2s. We will then quantify the increases in overhead and describe approaches we have taken to reduce this overhead. Finally, we will outline some possibilities for further improvements.

## 2. Concepts in Virtual Memory Management

Virtual memory is defined as a mapping, invisible to the programmer, between a process' virtual address space and the physical resources that contain the *data* of the address space. On most modern machines, this mapping is done using two layers of address translation: segmentation and paging. In segmentation, a virtual address is an encoding of a segment number and an offset within that segment. The physical address of the data is found by looking at a *hardware segment table*, which gives the location of the segment and its size. Address decoding for paging is similar to decoding for segmentation except that pages are fixed-sized whereas segments are variable sized. Figure 1 shows an example of how a virtual address is decoded into segments and pages<sup>1</sup> using the memory management unit of the AT&T WE<sup>®</sup> 32100 processor<sup>[1]</sup>.

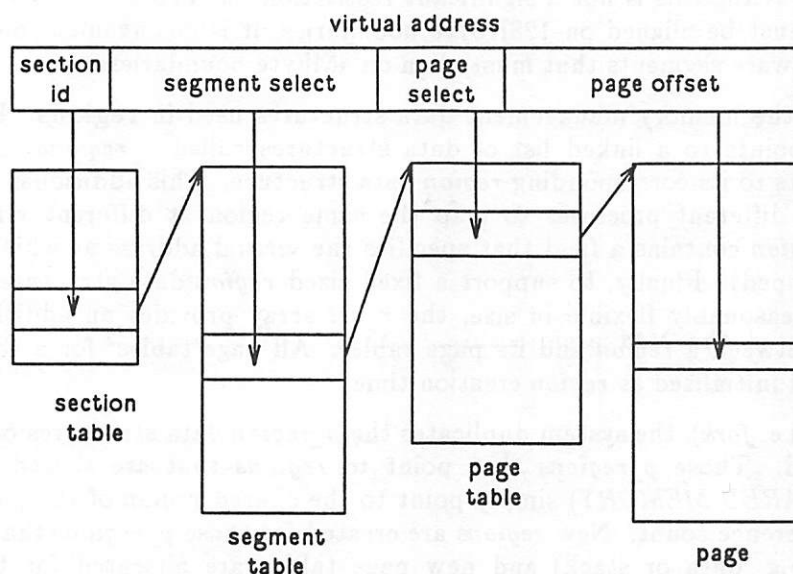


Figure 1. Decoding a Virtual Address on WE<sup>®</sup> 32100

When supported by adequate hardware, these address translation schemes can be used to provide both protection from other processes and improved usage of physical resources. For example, with the ability to map virtual pages to physical page frames, one can load virtually contiguous objects into non-physically contiguous pages. This eliminates the problem of fragmentation of physical memory. If demand paging is implemented along with a page replacement policy, one can support the execution of processes with virtual address spaces larger than physical memory.

### 3. The Regions Architecture

Before Release 2.1, UNIX System V did not provide a demand paging environment for the execution of large processes. The **regions** architecture for virtual memory was designed to provide this feature in a modular and portable way<sup>[2]</sup>. At its basis are the ideas that processes' address spaces consist of non-overlapping *regions* of virtually contiguous address ranges and that the data contained in these *regions* might be shared by multiple processes. Candidates for mapping into *regions* include the text and data sections of an a.out file, the process' stack area, shared library sections, and shared memory. Correspondingly, each *region* has a *type* that is either *PRIVATE* (for data), *SHARED TEXT*, or *SHARED MEMORY*.

As we shall see later, an important feature of the **regions** architecture is the association of page tables with these *regions*. This allows the sharing of *regions* and their data structures (e.g. page tables) but restricts *regions* to be aligned on hardware

1. On the 32100, virtual addresses are also grouped into one of four *sections*. Sections 0 and 1 are used for kernel address space, section 2 is used for the user text and data areas, and section 3 is used for the u-area and the stack. Sections allow address spaces to be sparsely defined without wasting too much space on segment tables.



segment boundaries. While this is not a significant restriction on AT&T's 3B2s whose hardware segments must be aligned on 128Kbyte boundaries, it is inconvenient on the 6386, which uses hardware segments that must align on 4Mbyte boundaries.

In Figure 2 we show the memory management data structures used in *regions*. Each process table entry points to a linked list of data structures called *p\_regions*. Each *p\_region* in turn points to its corresponding *region* data structure. This additional level of indirection allows different processes to map the same region at different virtual addresses (each *p\_region* contains a field that specifies the virtual address at which its *region* should be mapped). Finally, to support a fixed sized *region* data structure but allow *regions* to be reasonably flexible in size, the *r\_list* array provides an additional level of indirection between a *region* and its page tables. All page tables<sup>2</sup> for a *region* must be allocated and initialized at *region* creation time.

On process creation (i.e. *fork*), the system duplicates the *p\_region* data structures of the parent into the child. Those *p\_regions* that point to *regions* that are shared (e.g. *SHARED TEXT*, *SHARED MEMORY*) simply point to the shared *region* of the parent and increment its reference count. New *regions* are created for those *p\_regions* that are of type *PRIVATE* (e.g. data or stack) and new page tables are allocated for these *regions*. These page tables are initialized by copying the page tables of the corresponding *region* in the parent process. To avoid having to copy the pages of the private regions (e.g. data and stack), the *regions* implementation uses a copy-on-write (COW) technique. Those *PRIVATE region* page table entries that have been copied into the child use the same physical page frames as the parent but are set to "fault on write" in both the parent and child. Any attempt to modify the page by either the parent or the child will result in a protection fault. The protection fault handler then does the work of making a private copy of the page for the faulting process and mapping that private copy of the page into the writing process' address space.

The *regions* architecture provides a demand paging feature using page tables and disk block descriptors (DBDs). If a page is in memory, its page table entry describes its state and location. If a page is out on disk, its DBD entry provides similar state and location information. By providing *regions* as the unit of sharing of address space contents for processes, it is possible to implement such things as shared text and shared memory. Note that it is also possible to use the DBD to support memory mapped files although this has never been implemented. However, since the DBD is only designed to specify a disk location for the page, it cannot support the mapping of arbitrary devices into process address spaces.

#### 4. The VM Architecture

SunOS's original demand paging feature was based on one inherited from BSD 4.2. However, that implementation was considered *ad hoc* and non-portable. Sun's VM architecture is designed to improve the modularity and portability of the demand paging feature and to enhance the capability of the system in handling process address

---

2. In addition to hardware page tables, the *regions* architecture defines an auxiliary data structure called a disk block descriptor (DBD). Page tables and DBDs are always allocated in pairs and a page table entry's corresponding DBD entry can always be found a fixed number of words from the page table entry. The DBD can be viewed as a software extension of hardware page tables.

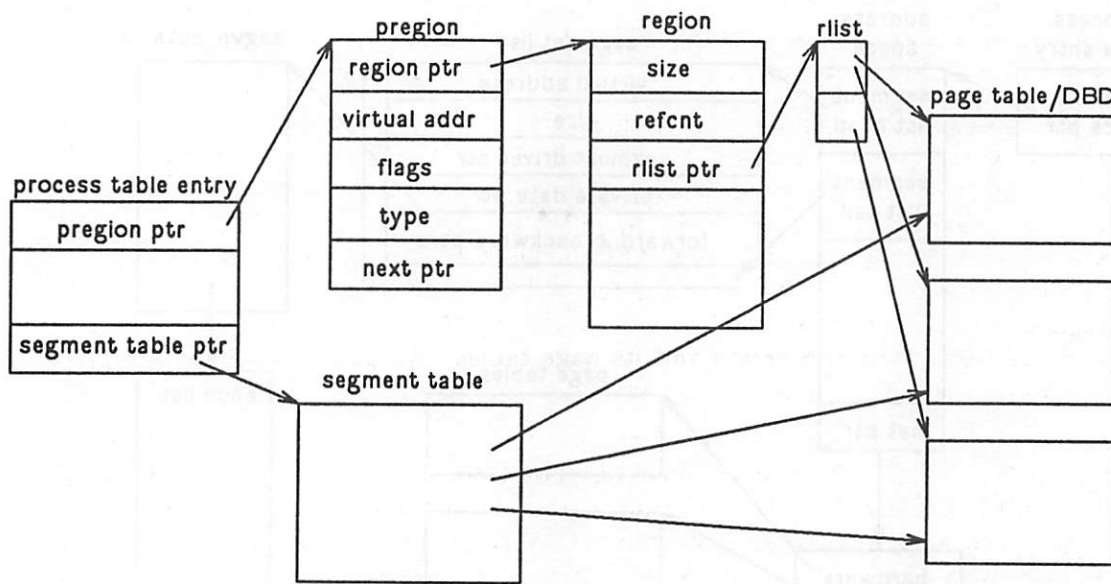


Figure 2. Regions Architecture

spaces and memory management.<sup>[3]</sup>

VM manages sparse process address spaces in a way similar to **regions**. Under VM, each contiguous range of virtual address space has a *segment*<sup>3</sup> data structure associated with it. Unlike **regions**, however, each *segment* is managed in an object-oriented manner. Each *segment* contains pointers to the functions (called *segment drivers*) that are allowed to operate on it. Different *segment* types will have different *segment drivers*. Also, unlike **regions**, page tables are not associated with *segments* but instead are associated with process *address spaces* (also an object/data structure under VM). This allows the details of the hardware memory management structure to be hidden from the *segment* layer. This has two major consequences: *segments* can begin on arbitrary page boundaries<sup>4</sup> instead of hardware segment boundaries, and the page tables of *segments* cannot be shared under VM because it is possible for two *segments* within a process' address space to use the same hardware page table. If two or more address spaces were to share that page table, they would all be required to contain both *segments*.

In Figure 3 we show the major data structures relating to process address space and memory management under VM. Each process has an address space data structure associated with it. This address space consists of two major parts: a hardware address translation (*hat*) data structure and a list of segments. The *hat* data structure is managed by the *hat* layer, which hides the details of the hardware memory

3. We have found this term to be a confusing one since there are hardware segments and VM segments. Henceforth, throughout this paper, *segments* will be used to refer to VM segments unless we explicitly use the term "hardware segment."

4. Note that this page alignment requirement still requires the *segment* layer to know one detail of the MMU: the page size.

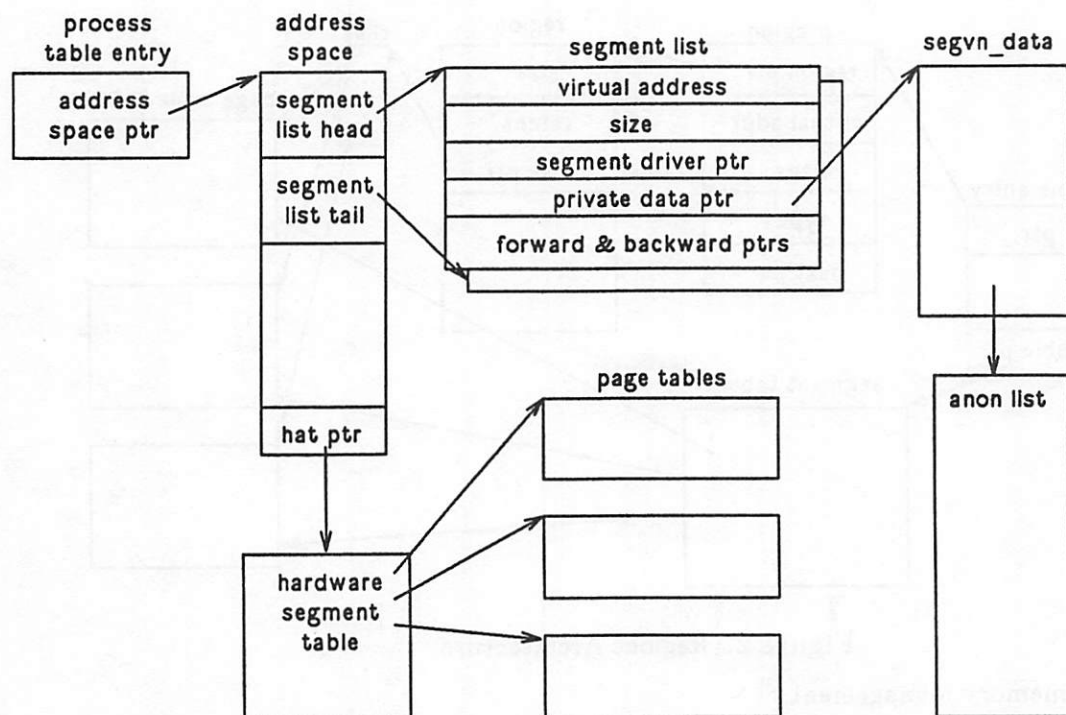


Figure 3. VM Architecture

management unit. Each *segment* describes a contiguous virtual address range. For example, one segment might describe the text section for a process. Another might be a memory-mapped file. Still another might describe a frame buffer for managing a bit-mapped display. Since each *segment* type could involve different operations for managing faults, the page fault handler uses the *segment* drivers of each *segment* to service page faults.

An important feature of VM is the integration of the system buffer cache with the system-wide page cache<sup>5</sup>. In the past, the system set aside an area of memory that was used as a buffer cache for filesystem I/O. The size of each buffer was matched to the size of the filesystem block and was indexed in the cache by an *(inode, block number)* pair. Under VM, each physical page frame in the system is now potentially a page-sized I/O buffer that is part a system-wide page cache, indexed by a *(vnode, page number)* pair, where the *vnode* is the filesystem-independent representation of the file<sup>[4]</sup>. Every page that has a current copy of data from the filesystem can be found via its *(vnode, page number)* pair. Pages that do not have an identity on a filesystem (e.g. data pages that have been modified or stack pages) are called anonymous pages and are artificially given an association with a *(swap vnode, page number)* pair.

Because it is possible to find any incore page via its *(vnode, page number)* pair, VM allocates and initializes page table entries at fault time rather than at *segment*

5. Actually, there is still a small system buffer cache but its use is now limited to reading superblocks, indirect blocks, and inodes.

initialization time. Thus both address space ranges and page tables are allocated sparsely under VM (under **regions**, address space ranges are allocated sparsely but page tables are not).

### 5. Performance Analysis

We ran numerous benchmarks to compare the performance of UNIX System V Release 3.2 (**regions**-based) with development loads of UNIX System V Release 4.0 (VM-based). Of relevance to our discussion here are system call benchmarks that measure *fork* and *exec* performance and a multi-user benchmark<sup>[5]</sup> that measures system throughput. In Table 1 we compare the cost of *fork* and *exec* for the two operating systems.

TABLE 1. CPU Cost for *fork* and *exec*

	regions	VM
<i>fork</i>	31.7 ms	57.2 ms
<i>exec</i>	34.9 ms	53.1 ms

We can see that under VM, the costs of *fork* and *exec* increased significantly. This increase in cost was reflected in the performance of the multi-user benchmark, which is dominated by many *fork*/*execs* of small, short-lived processes. Examination of system activity data (via *sar(1)*<sup>[6]</sup>) collected during the benchmark run also led to an interesting observation: the number of page faults incurred by the benchmark under VM was significantly higher than under **regions**. In Table 2, we compare the number of page faults incurred by the benchmark under the **regions**-based and VM-based systems.

TABLE 2. Number of Faults Under **regions** and VM

	regions	VM
valid faults	1172	3040
protection faults	1306	1098

The reasons for this were not immediately obvious since the same a.out files were used to execute the benchmark under both releases and, at the multiprogramming levels being examined, there was no page stealing. We finally discovered that this was occurring because VM did not allocate and initialize page tables on process creation (i.e. *fork* or *exec*). Recall that under **regions** page tables **had** to be initialized on process creation. Thus, under **regions**, all pages that were valid for the parent became valid for the child after a *fork*, and *execs* would attach shared **regions** and their page tables to the child. Under VM, no pages are valid for a process after either a *fork* or an *exec*. It is through page faults that the page tables become initialized and it is these additional page faults that account for the additional page faults seen under VM. The extra CPU costs of *fork* and *exec* are partially explained by the extra faults that the new process incurs after these system calls.

To gauge how much these extra faults cost in system CPU time, we used the CASPER instrumentation and tracing package<sup>[7]</sup> to obtain high resolution timing measurements of the CPU requirements of the validity fault handler and the protection fault handler. Table 3 lists the CPU costs for two types of faults under the two operating systems: validity faults that find a page incore (i.e. no I/O needed), and protection faults.



TABLE 3. CPU Costs of Faults

type	regions	VM
valid fault (no I/O)	1.9 ms	3.5 ms
protection fault	2.6 ms	4.3 ms

To investigate the extra costs for doing page fault handling under VM, we use the function trace facility of CASPER to instrument all the functions that were part of the VM module. We found that function call overhead to functions in the VM module accounted for 20% of the CPU costs of handling a validity fault. Not wishing to violate the object oriented structure of VM, we decided to concentrate our efforts in improving performance by reducing the number of page faults incurred by processes.

#### 6. Reducing Page Faults on fork

Presumably the reason for not initializing the page tables of children on *fork* is not to spend time and resources constructing page tables for a process that is going to *exec* soon. However, the cost of building page tables for small processes is negligible compared to the cost of servicing a page fault even if the page is in memory (see Table 3). Two other points are important here: there is always some user level code that must be executed between returning from a *fork* and executing the *exec*, and most processes that *fork* are relatively small shells (e.g. Bourne shell, Korn shell, *csh*), which do a non-trivial amount of work between *fork* and *exec*. Therefore, in practice, the work in copying and initializing page tables is not high compared to the cost of handling even a single page fault.

By simply allocating page tables and copying page table entries for the child process on *fork*, we can save a tremendous number of page faults when executing commands through a shell. In Table 4 we compare the number of validity faults and protection faults needed to execute one script of our multi-user benchmark before and after this small change in *fork*.

TABLE 4. Number of Faults when Duplicating Page Tables on *fork*

	VM	VM with Dup
valid faults	3040	1116
protection faults	1098	1273

#### 7. Reducing Page Faults on exec

Under **regions**, it is possible for multiple processes to share *regions* and their page tables (and therefore the physical page frames referred to by the page tables). If one incore process faults and validates a page in memory, that page becomes valid (i.e. no page fault on reference) for all processes sharing that *region*. Under VM, the sharing occurs at the physical page frame level, not the page table level. Page faults are used to propagate sharing information between page tables.

Given the high cost of page faults under VM, it is desirable to simulate the sharing of page tables by initializing page table entries for *segments* on *exec*. This is simple to implement because each vnode has a pointer to a list of incore pages associated with that vnode. On *exec*, we go through the *segment* list associated with the process' address space. For each *segment*, we check to see if it has a vnode mapped to it. If it does, we identify the vnode and walk through the list of pages associated with that vnode. Since we know the virtual address at which the *segment* is mapped, we can find



the page table entry corresponding to each page in the list. We then initialize the page table entry that corresponds to that file page, allocating a page table if needed.

In a sense, we are estimating the working set of the process after *exec* using the set of incore pages associated with the vnodes used by the *execing* process. In Table 5, we show the savings in page faults over the original VM implementation while running the multi-user benchmark.

TABLE 5. Number of Faults when Building Page Tables on *exec*

	VM	VM with Dup and Build
valid faults	3040	840
protection faults	1098	1122

### 8. Reducing Copy-on-Write (COW) Faults

In the section on the *regions* architecture, we described the rationale behind the implementation of copy-on-write (COW) on *fork*. A similar idea is used in VM to avoid having to copy all the pages of the private *segments* on *fork*. Unfortunately, under VM, the cost of handling a COW fault is more than four times the cost of copying a page.<sup>6</sup> This means that if more than 1/4 of the incore pages of a *segment* are modified after a *fork*, it would have been better to copy the incore pages of the *segment* instead of deferring the copy.<sup>7</sup> Ideally, we would like to avoid COW faults without unnecessary copying. The question becomes: can we know which pages of a process will be modified after a *fork*? In general we cannot, but if we examine the uses of *fork* and *exec* in typical programs, we can make reasonable guesses.

For example, we know that most programmers write their *fork* statements as follows:

```
if ((pid = fork()) {
    .
    .
}
```

If *pid* is not a register variable, we know that this statement will result in COW faults in both the parent and child on either a data page or a stack page. Furthermore, if *pid* is a variable that is local to the function invoking the *fork*, then we would know that the COW faults for changing *pid* will occur on a stack page. By looking at the user stack pointer register, we can determine exactly the page the COW fault will occur on and do the copy to avoid those two COW faults (one in child and one in parent). CASPER instrumentation showed that, in practice, the page pointed to by the user stack pointer always incurred a COW fault after a *fork*. Using this technique to find the correct stack page to copy on *fork* could yield a savings of two COW faults per *fork*.

However, there is a more general approach that yields much larger savings in COW faults. The basis of this approach, which we have termed *bovophobic*<sup>8</sup>, again comes

6. In Table 3, we see that the cost of a protection fault, which handles the COW fault, is over 4 ms. Our measurements also showed that it takes about 1 ms to copy a page. Hence the 4:1 ratio above.

7. Actually, since the parent is also set to fault on write to those pages, the fraction becomes more like 1/7.

8. Fear of COWs.

from an analysis of how *fork* and *exec* are used. Recall our earlier observation that shells are the major invokers of *fork*, and these shells usually invoke *fork* to execute commands. It would be reasonable to assume that the execution paths taken by a shell each time a command is executed is the same. We can further assume that the variables that are modified after one command execution will be modified after another command execution. In terms of pages, we can expect that the pages that have COW faulted earlier will COW fault again. Again, CASPER instrumentation verified our prediction. For each shell program that we examined (Bourne shell, ksh, make, cc) the same pages are always modified between *fork* and *exec*. Thus, the set of pages that were modified after a previous *fork* is a reasonable predictor for the set of pages that will be modified after a future *fork*.

Fortunately, the VM implementation provides us with an approximation to the set of pages modified after a previous *fork*. Recall from our earlier description of VM that pages of a *segment* that have no identity on the filesystem are called anonymous pages and are associated with the swap vnode; these pages are kept on an *anon* list belonging to the *segment*. In order for a page to be on the anon list, it must have been modified at some point. In other words, the anon list contains a list of pages that have been modified by the process. This provides an approximation to the list of pages modified after a *fork* (it may also include pages modified after an *exec*). Thus on *fork*, we can walk through and copy all pages on the parent's anon lists for the child.<sup>9</sup> Table 6 compares the number of protection faults incurred before and after this change.

TABLE 6. Number of Faults when Copying Anon Pages on *fork*

	VM with Dup and Build	VM with Dup, Build, and Bovophobic
valid faults	840	640 <sup>10</sup>
protection faults	1122	142

## 9. Summary and Further Work

The initial implementation of VM resulted in an increase, compared to *regions*, in the number of page faults generated by processes. This, coupled with the high cost of page faults under VM, led to poor performance with respect to many of our benchmark loads. By duplicating a child's page tables on *fork* and initializing a process' page tables on *exec* to simulate shared *segments*, we have been able to reduce the number of page faults generated after process creation to a level comparable to the *regions*-based system. Furthermore, with our *bovophobic* technique of using the anon list of each *segment* to predict pages that will be modified after a *fork*, we have been able to significantly reduce the number of COW faults needed to execute commands.

9. To avoid unnecessary copying of pages that are on the anon list because they were modified after an *exec* but not after a *fork*, we could include a bit field in the anon structure to keep track of whether a page was entered in the anon list after a *fork* and/or *exec*. On *fork* we would only copy those pages that were marked "modified after *fork*." In practice, CASPER instrumentation has showed that only one extra page is copied for the Bourne shell and no extra pages are copied for ksh.

10. We were also able to achieve a reduction in valid faults by simply increasing the number of incore inodes for the system. For an explanation of this, see the next section.

During our studies of factors affecting the performance of the system, we found that the performance of the initial VM implementation was sensitive to the number of incore inodes in the system and to the size of the directory name lookup cache (both tunables). While measuring the performance of simulating *shared segments* under *exec* we found that the number of validity faults incurred by the system was also sensitive to this parameter. We realized that since our initialization of page tables on *exec* depended on walking through the incore page list associated with the vnode, inode reuse (and therefore vnode reuse) can result in the loss of access to those incore pages. That is, when an inode is reused, the association of each page on the incore page list for that inode is broken and that page can no longer be found in the page cache. On the original VM implementation, this meant that any faults on those pages would result in I/O to read those pages in even though they may still be incore. On our implementation, since we no longer have a complete incore page list to walk through to initialize the page tables, we incur more valid faults that also must do I/O to read those pages in. Our solution to this problem is presented elsewhere<sup>[8]</sup>.

We can also extend our idea of simulating the sharing provided by the *shared regions of regions* by implementing an equivalent *shared segment*. With a *shared segment*, page tables would be associated with this *segment* in addition to being associated with the hat layer of the address space, and the *shared segment* would have to be aligned on hardware segment boundaries, as was the case in *regions*. This would have the advantage of true sharing in which the first process faulting on a page in a *shared segment* would result in the page becoming valid for all processes attached to the *shared segment*. For architectures with small hardware segment sizes, this could be used as the default for the *text segment* while for architectures with large hardware segment sizes, this could be used to tradeoff fault performance with memory wastage and address space fragmentation.

Finally, we have concentrated our efforts on the simple mechanical overhead of paging in the absence of memory contention. We have not looked at issues of page stealing and working set management. Much work had originally been done to improve the management of memory contention under *regions*, and in the future, we hope to incorporate some of these ideas under VM.

#### 10. Acknowledgements

We were part of a team called in to bring Release 4 performance up to acceptable levels. Other members of this team included Bob Butera, Khosrow Dahi, Jerry Feder, Dick Menninger, Don Milos, and Hung Ping Wong. We would especially like to thank Jerry Feder for his leadership in this work, Bob Butera and Dick Menninger for pointing out the reasons for extra page faults under VM, and Don Milos for taking our prototype code and turning it into production quality for inclusion in Release 4.

#### REFERENCES

1. WE<sup>®</sup> 32100 Microprocessor Information Manual, AT&T Select Code 307-730, 1986.
2. M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1986.
3. R. A. Gingell, J. P. Moran, and W. A. Shannon, "Virtual Memory Architecture in SunOS," *Proc. of the 1987 Summer USENIX Conf.*, June 1987, pp. 81-94.

4. S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in SUN UNIX," *Proc. of the 1986 Summer USENIX Conf.*, June 1988, pp. 238-247.
5. S. Gaede, "A Scaling Technique for Comparing Interactive System Capacities," *Conf. Proc. of CMG XIII*, December 1982, pp. 62-67.
6. *UNIX System V: User's Reference Manual*, AT&T Select Code 307-232, 1986.
7. R. E. Barkley and D. Chen, "CASPER the Friendly Daemon," *Proc. of the 1988 Summer USENIX Conf.*, June 1988, pp. 251-260.
8. R. E. Barkley and T. Paul Lee, "A Dynamic File System Inode Allocation and Reclaim Policy," *Proc. of the 1990 Winter USENIX Conf.*, January 1990.



**Danny Chen**  
AT&T

Danny Chen has been working in Bell Labs on UNIX performance problems for almost 10 years. For the past 6 years, he has been with Bell Lab's Performance Analysis Department and helping the UNIX Developer's in Summit in solving kernel performance problems. He has an M.S. in Computer Science from U.C. Berkeley, a B.S. in Computer Science and a B.A. in Physics from Columbia University, and graduated from Stuyvesant High School in New York.



**T. Paul Lee**  
AT&T

T. Paul Lee joined Bell Labs in 1984 after earning his Ph.D. degree in computer science from the University of California at Berkeley. At Bell Labs, Paul was active in the performance analysis of the UNIX operating system, most recently working on performance improvements for UNIX System V Release 4.0. He is currently employed at Sony Microsystems in San Jose, California. Before his graduate work at Berkeley, Paul worked for Motorola and Bell Northern Research in the area of microprocessor design and firmware programming.



**Ronald E. Barkley**  
AT&T

Ronald Barkley has been with AT&T Bell Laboratories since 1984. As a member of the UNIX System Development Department, Ron was involved in the performance analysis and improvement of System V Releases 3 and 4, especially in the area of memory management. He is currently working on the analysis of AT&T's implementation of the OSI protocol stack. Ron has a MS degree in mathematics from Clemson University and a MS degree in computer science from Rutgers.

**TAE Plus: Transportable Applications Environment  
Plus a User Interface Development Tool for  
Building Graphic Oriented Applications**

*Martha Szczur  
Karl R. Wolf  
NASA/Goddard Space Flight Center*

The text for this paper is at page 363.





# Implementing a Mach Debugger For Multithreaded Applications

Deborah Caswell  
Hewlett Packard Laboratories

David Black  
Carnegie Mellon University

November 1, 1989

## Abstract

Multiple threads of control add new challenges to the task of application debugging, and require the development of new debuggers to meet these challenges. This paper describes the design and implementation of modifications to an existing debugger (gdb) for debugging multithreaded applications under the Mach operating system. It also describes the operating system facilities that support it. Although certain implementation details are specific to Mach, the underlying design principles are applicable to other systems that support threads in a Unix<sup>1</sup> compatible environment.

## 1 Introduction

Applications that employ multiple threads of control add new challenges to the debugging problem that exceed the capabilities of most existing debuggers for single threaded applications. Not only do the debuggers lack the logic and data structures needed to deal with multiple threads in the target application, but often the operating system primitives they depend on (e.g. *ptrace*) are inadequate to the task. While a good operating system designer includes new primitives to support multithreaded debugging, it is up to the debugger implementors to put these primitives to good use. This paper describes one such effort in this area; the modification of an existing debugger for single threaded Unix applications to support debugging of multithreaded applications under the Mach operating system. Although the operating system primitives are specific to Mach, the underlying design principles are applicable to any debugger for applications with multiple threads of control in a Unix-compatible environment.

This paper begins with an introduction to Unix debuggers for single threaded applications and the problems involved in adding support for multithreaded applications. We then address these problems and describe our solutions for the Mach debugger that we have implemented. We conclude with a short discussion of possible extensions to our work and its relation to other work in the field.

---

<sup>1</sup>Unix is a registered trademark of AT&T Bell Laboratories

## 2 Unix Debuggers and Debugging-Related System Calls

On a Unix system, the debugger and an application being debugged are separate user processes that communicate via kernel-provided services, primarily the various options of the *ptrace* system call. In addition a debugged process undergoes special interactions with the Unix signal and process control mechanisms to facilitate debugging. Hence the implementation of debugging functionality is split across the user-kernel boundary and includes logic in both the debugger and the operating system.

### 2.1 *ptrace*

The *ptrace* system call implements most of the functionality required by a debugger to interact with the application being debugged. The various options to *ptrace* implement:

- Access to the application's registers and memory.
- Read access to certain kernel information about the application (e.g. the command it was invoked with).
- Continue or single step the application (with the option of delivering a signal). Also kill (terminate) the application.

Since Unix applications do not have access to the state or address space of another process, even when running in kernel mode, *ptrace* is implemented by having the debugged application perform the desired action on itself (e.g. if a register is to be read, the debugged application reads the register and returns that value to the debugger process via an internal kernel communication mechanism); the kernel code that the application executes to do this is in the function *procxmt*. The debugger process sleeps waiting for the application to wake it when the action has been completed.

*ptrace* also includes a hook that allows the debugged application to identify itself to the kernel and request special treatment as a result. Typically a debugger forks a child process which calls this hook ('Please trace me') before *exec*'ing the application to be debugged; this hook sets the trace flag (STRC) for the debugged process. A process with this flag set is referred to as a "traced" process.

### 2.2 Signals

Signals form an important part of the interface between the debugger and the debugged application. Signals are caused both by external events involving the application and by exceptions (both hardware and software) occurring in the application itself. When a signal is delivered to a traced process, the process stops inside the signal code so that it can carry out actions requested by the debugger through the *ptrace* interface. The process remains in this state until a continue, single step, or kill action is requested; the first two allow it to

continue execution, and the latter terminates it immediately. The signal itself is cancelled unless the debugger specifically directs that it be taken; the debugger may also substitute another signal if desired.

The two most common uses of signals during debugging are breakpoint/trace detection and interruption of a looping application. Both breakpoints and trace traps generate the signal SIGTRAP; the debugger will usually allow this signal to be cancelled after gaining control via *ptrace*. Similarly, interruption characters at the keyboard also generate signals (e.g. control-C generates SIGINT). A user can take advantage of this to regain debugger control of a runaway application; typing control-C on the terminal causes the application to stop in the signal code and the debugger to regain control. The only remaining detail to be explained is how the debugger detects that the application has stopped.

### 2.3 *wait*

*wait* is the most common mechanism used by debuggers to determine that a debugged application has stopped<sup>2</sup>. The *wait* system call is normally used to wait for an exited process and return its exit code, but *wait* also returns if it finds a traced process that has stopped in the signal code. Thus a debugger can use *wait* to simultaneously detect the two occurrences that it is most interested in:

1. The application has taken a signal (e.g. hit a breakpoint) and stopped.
2. The application has exited.

The status returned by *wait* indicates whether the application has stopped or exited, and in the former case also returns the signal that caused it to stop. Once *wait* has indicated that the debugged application has stopped, the debugger can safely use *ptrace* to manipulate the application.

### 2.4 Breakpoints and Single Stepping

The actual mechanisms used to implement breakpoints and single stepping are hardware dependent, but the software interface to them (via *ptrace*) is very similar across most machines. A breakpoint is usually implemented by a special machine instruction which causes a trap that the kernel immediately identifies as a breakpoint trap. Similarly, single stepping is often implemented by a 'trace bit' which causes a trace trap at the end of the next instruction. In both cases the kernel identifies the trap and generates the SIGTRAP signal; this notifies the debugger via the kernel mechanisms discussed above.

There is one subtlety involved in actually using breakpoints. Since it is not possible to insert the special breakpoint instruction between adjacent instructions, it is necessary to

---

<sup>2</sup>Used by virtually all BSD debuggers; some System 5 debuggers may use the SIGCLD signal for this purpose.

replace part of the application's machine code<sup>3</sup>. As a result, it is necessary to execute the replaced code when continuing from a breakpoint; this is usually done by replacing the original code, single stepping, and replacing the breakpoint before continuing the application.

### 3 Multiple Threads and Debugging

Mach splits the familiar Unix **process** abstraction into the **task** and **thread** abstractions [9]. A **task** is an address space along with associated communication rights; it is a passive entity that is not capable of execution by itself. A **thread** is a locus of control within a task; it consists of a register context, a kernel stack, and usually its own user stack (assigned under user control). A Unix process is represented by a task with a single thread under Mach. Additional threads can be created within a task for concurrency and parallelism; the threads share all of the task's resources (memory and communication rights). Threads execute and are scheduled independently by the operating system kernel, so that a blocking operation executed by one thread does not affect others. A debugger normally executes in a task distinct from the application that is being debugged for protection reasons.

Using existing process-oriented debuggers to debug applications with multiple threads exposes a number of problem areas in both the debuggers and the underlying operating system support. Among the more significant of these problems are:

- The debugger lacks internal data structures and process control logic for tracking and controlling the states of the application threads.
- The existing *ptrace* interface and process control logic are inadequate because they assume that there is exactly one application thread.
- Concurrent exceptions (e.g. three threads hit a breakpoint before the debugger gets control) are beyond the ability of both the debugger and the operating system. The exceptions will be serialized, causing potentially bizarre behavior (e.g. a thread appears to hit a breakpoint that was just removed).

We attacked these problem areas individually by improving *ptrace* to deal with multiple threads, replacing the signal and *ptrace* logic that serializes concurrent exceptions, and enhancing the debugger to understand multiple threads and concurrent exceptions. The following sections discuss our implementation of this strategy.

### 4 Improving *ptrace*

Our major goal in improving the implementation of *ptrace* was to eliminate *procxmt* and the associated logic that resumes the debugged application. The interaction of *ptrace* with

---

<sup>3</sup>This is not strictly true for pipelined machines with address queues that allow out of order instruction sequencing (e.g. Hewlett Packard's Precision Architecture). The breakpoint and trace trap mechanisms for such machines can be rather complex.



signals requires that the entire application be resumed, with the result that the threads not stopped in the signal code continue to execute, possibly altering state needed for the debugging process. The new implementation takes advantage of existing Mach features to replace the *procxmt* code (executed by the application to perform actions on itself) with code that allows the debugger process to directly perform the actions itself. This has the beneficial side effect of eliminating two context switches and the associated synchronization from every call to *ptrace*. The following sections describe these changes and also indicate how the *ptrace* actions could be replaced by other Mach kernel facilities available to a debugger.

#### 4.1 Access to Memory and Registers

The *ptrace* actions that read or write the application memory can be replaced by the Mach *vm.read* and *vm.write* kernel operations that directly access memory of another task. These operations are somewhat inefficient in accessing the small amount of data that *ptrace* normally transfers. This inefficiency can be alleviated by directly using *vm.read* and *vm.write* in the debugger to examine or write larger amounts of data, or by system-specific extensions to *ptrace* for transferring larger amounts of data; in both cases the number of kernel operations required to access large amounts of data is reduced.

Similarly, the *ptrace* actions that read or write the application registers can be replaced by the Mach *thread.get.state* and *thread.set.state* kernel operations which also read and write registers. Since user registers are saved on the kernel stack during kernel operations, this improvement is made possible by the fact that Mach places the kernel stacks of threads in kernel memory proper, so that all stacks are accessible to the kernel at all times (in contrast Unix usually forbids access to kernel stacks of other than the current process).

#### 4.2 Other Kernel Information

The *ptrace* action that reads kernel information from the *user* area in Unix indicates which information it wants by an offset into the user structure. Since the user structure has been divided into task and thread specific components in Mach, it was necessary to incorporate kernel code to reassemble a fake user structure from these components so that the offset can be interpreted to find the desired data. This information is also available via the various options to the Mach *table* syscall.

#### 4.3 Execution Control

The *ptrace* continue action can be replaced by the Mach *task.resume* kernel call. Finer control over execution can be obtained by using *thread.suspend* and *thread.resume* to control which application threads are to execute. Single stepping can be implemented by accessing the process control register(s) of the desired thread(s) to set the trace bit (as described in the previous section) before performing the *task.resume*.

There is, however, no direct Mach replacement for the termination action of *ptrace* due to the need to clean up and deallocate Unix state and data structures. It is still necessary for the application to perform this itself by calling *exit* either within or from outside the kernel. The debugger can cause the application to do this reliably by sending SIGKILL to it.

#### 4.4 Mach ptrace

We have applied the above changes to the Unix implementation of *ptrace* to yield the Mach implementation. The resulting implementation does not resume the debugged application in order to examine or modify its state. Although the above changes have been described in terms of the calls that could be made by a debugger outside the kernel, we have taken advantage of the efficiencies afforded by an in-kernel implementation. (e.g. modify only the register requested, rather than performing a *thread\_set\_state* to modify all of them.). The resulting *ptrace* makes it possible to use existing single threaded debuggers for multithreaded applications because it does not resume the application in order to examine or modify its state. This does not constitute complete support for multithreaded debugging due to lack of support for concurrent exceptions<sup>4</sup>, but it is a distinct improvement over the original implementation which is essentially useless for this purpose.

### 5 Handling Concurrent Exceptions

The interaction of *ptrace* with signals poses a major obstacle to multithreaded debugging because signal semantics require that exceptions be handled serially. This is a reasonable restriction for external event signals and a single-threaded application, but it is unreasonable for exception signals generated by a multithreaded application because it hides some of the application's state. In particular, it is impossible for a debugger to determine the complete state of a multithreaded application that has hit multiple breakpoints in different threads because only one breakpoint can be reported to the debugger. If the user were to remove some of the other breakpoints before continuing the application, some of the pending breakpoint signals would appear to come from non-existent breakpoints. This and similar problems are a direct result of the serialization imposed by signals and can only be alleviated by using a different mechanism to report exceptions to the debugger.

#### 5.1 The Mach Exception Handling Facility

Mach provides a message-based exception handling facility to overcome the limitations imposed by Unix signals. Converting exceptions to messages allows a debugger to receive and record all of the exceptions that have happened to the application since it was last

---

<sup>4</sup>This also causes problems with single stepping in the presence of outstanding breakpoint events because the signal received after initiating the single step action may be from a pending breakpoint instead of completion of the single step.

continued, and thus obtain a complete picture of the application's state. The exception handling facility is based on a remote procedure call (rpc) implemented by two Mach ipc messages. The occurrence of an exception causes a message to be sent that identifies the exception, the thread that caused it, and the task containing that thread. The thread waits for a reply to this message; the reply indicates whether the exception was successfully handled. Further details on the use and implementation of the exception handling facility can be found in [4].

A debugger uses the exception handling facility to detect exceptions in the debugged application, including breakpoint and trace traps. The debugger enables this by setting the **task exception port** of the debugged task to a port on which the debugger expects to receive exception messages. Then any exception in the application that is not handled by an application-specific error handler initiates an exception rpc to the debugger. This mechanism replaces signals and the serialization restriction imposed by them; multiple concurrent exceptions result in multiple messages being queued for the debugger. The debugger can dequeue and examine all of these messages without continuing the application and thus determine the complete state of the application, including which threads are at breakpoints.

## 5.2 Unix Compatibility

Unix compatibility is provided by an internal kernel exception handler (the `ux_handler`) that converts exception messages to signals and sends them to the appropriate threads. This handler sets up its port as the task exception port of `/etc/init`; the inheritance of the task exception port across task creation (including forks) ensures that exceptions are converted to signals by default unless there is an application that is interested in handling the exceptions, *e.g.* a debugger. A debugger can use the `ux_handler` to convert exceptions to signals for debugging purposes; forwarding an initial exception message or sending a new one to the `ux_handler` causes this to occur. The `ux_handler`'s port can be obtained as the original task exception port of the application before the debugger changed it.

The signal compatibility code in the Mach kernel has been extended to ensure that signals generated by exceptions are handled by the thread that caused the exception. This behavior is applied to the nine Unix signals caused by exceptions; `SIGILL`, `SIGTRAP`, `SIGIOT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, and `SIGPIPE`. As a result, the `ux_handler` can be assured that the thread to which it delivers an exception signal (*i.e.* the thread that caused the original exception) will handle the signal.

## 6 A Multithreaded Debugger

We have developed a debugger for multithreaded applications that utilizes the Mach kernel's debugging support. This debugger is an enhanced version of *gdb* from the Free Software Foundation [8]. This section describes the enhancements and the additional functionality they provide.

## 6.1 User Interface Extensions

We extended *gdb*'s user interface to allow the independent examination and modification of threads in the application. New commands were added to identify threads and to select a thread to manipulate. Our debugger adopts the concept of a *current thread* to simplify the interface and maintain maximum compatibility with the single threaded version of *gdb*. Although our enhanced *gdb* can only manipulate one thread at a time, it allows the user to select any thread as the *current thread* to be manipulated. Hence the same *gdb* commands that modified the state of a Unix process perform the same modifications to the selected thread. We also changed the single step command to single step only the current thread while preventing all other application threads from running.

In order to isolate aberrant behavior such as race conditions, it is vital to be able to continue only selected portions of a multithreaded application. To support this we also added debugger commands to individually suspend and resume threads. A *suspended* thread does not execute when the application is continued, and remains in this *suspended* state until it is *resumed*. These commands also support options to suspend or resume all threads to allow faster selection of the desired portion of the application (e.g. one can select two threads out of a collection by suspending all of the threads and then resuming the two desired threads). The debugger can resume a thread, but the thread remains suspended until the entire application containing the thread is continued.

## 6.2 Debugging and Concurrent Events

The major new complication posed by debugging multithreaded programs is that instead of one event (a breakpoint), many events can happen concurrently when the application is continued. Due to parallelism and concurrency within the operating system itself, these events may not be reported to the debugger in chronological order (e.g. two threads hit breakpoints on a parallel processor, but a clock interrupt causes the first thread to be delayed in preparing its exception message so that the second exception message is sent first). In particular, the Mach exception facility makes no guarantees that messages will be delivered in the order that the exceptions occurred. In most cases the order of the events is not of major interest; for the case of breakpoints, the user is probably more interested in the fact that two threads just hit breakpoints (and which breakpoints they hit), than in which breakpoint occurred first.

## 6.3 Managing Thread Execution

Our debugger uses a six state model to manage the execution states of threads in the application it is debugging. This model underlies the debugger logic for dealing with concurrent events. The six possible execution states for a thread being managed by our debugger and the associated state transitions are shown in Figure 1. The reader should note that the Mach suspend and resume mechanisms are based on reference counts (e.g. a thread that is suspended twice must be resumed twice) and that the task and thread suspension mechanisms



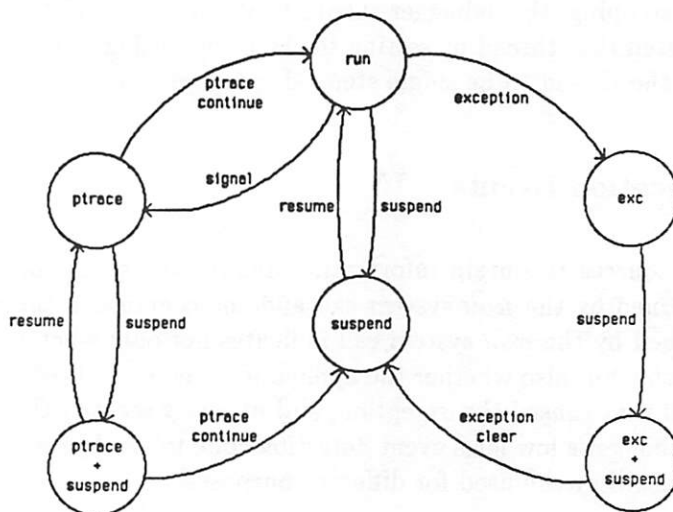


Figure 1: Application threads state diagram

are independent (e.g. the effects of a *task\_suspend* cannot be reversed by a *thread\_resume*). A thread may run only if both it and its task are not suspended. For clarity, only the initial suspend (either task or thread) that makes a thread not runnable, and the final resume that makes the thread runnable, are shown in Figure 1. No other suspends and resumes cause state transitions.

With this state diagram, we can now explain how the debugger manages thread execution. All of the threads for a running application are in the **run** state; events that invoke the debugger cause one of two possible transitions out of this state:

1. The event is an exception, causing the thread to send a message to the debugger. The thread waits in the **exc** state for a reply.
2. The event is the delivery of a Unix signal. The thread is stopped by the Unix signal code and in the **ptrace** state where it can be manipulated using *ptrace*.

In response to the first exception detected (from a set of concurrent exceptions), the entire application is suspended, causing threads in the **run** and **exc** states to move down to the corresponding **suspend** states. The debugger proceeds to obtain information about the remaining exception events and clears the exceptions by sending reply messages (moving the corresponding threads from the **exc + suspend** state to the **suspend** state). The debugger is now prepared to interact with the user, with all of the application threads in either the **ptrace** or **suspend** states. The Unix signal logic guarantees that at most one application thread can be in the **ptrace** state.

When the user requests that the application be continued, the debugger continues the application by using either *task\_resume* (if the debugger suspended the task) or the continue action of *ptrace* (if the kernel ptrace code suspended the task). Threads individually



suspended by the user remain in the **suspend** state; all others enter the **run** state and resume execution. For single stepping, the debugger suspends all but the thread to be single stepped, arranges to single step that thread by setting the trace bit (using *thread\_set\_state*), and resumes the task; only the thread to be single stepped enters the **run** state.

## 6.4 Processing Application Events

Our debugger utilizes two sources to obtain information about events that occur in the application, the status returned by the *wait* system call and the contents of the exception messages. The status returned by the *wait* system call indicates not only whether a thread has stopped in *ptrace*, and why, but also whether the application has terminated. Exception messages identify the thread that caused the exception, and precisely identify the exception as well. This enables our debugger's low level event detection code to treat breakpoints and trace traps differently because they are used for different purposes.

Trace traps need not be reported directly to the user by the low-level event detection code because they are used for exactly two purposes: user-requested single stepping, and debugger-initiated single stepping as part of continuing from a breakpoint. In both cases higher-level debugger code can figure out if the user should be notified and how, so the events are passed directly to that code without reporting the trap directly to the user. The higher level debugger logic will return control to the user or continue the application as appropriate. In the second case, the user is not notified about the internal debugger actions required to continue from a breakpoint. Similar reasoning applies to breakpoints used to implement stepping over functions, subroutines, or procedures as part of a language-level (e.g. C) single step.

In contrast, breakpoint traps caused by user-requested breakpoints are reported directly to the user by the the low-level event detection code. This is so the user is aware of which threads have hit breakpoints since the last time the application was continued. The only breakpoints not reported to users are those used as part of a language-level single-step as indicated above.

Our debugger must use a polling low-level event detection algorithm due to the incompatibility of the sources of event information. The polling interval is implemented by a timeout if no message is received within the timeout interval, along with the use of a nonblocking variant of *wait*. Upon detecting an event, the algorithm loops to obtain information about all pending events; this guarantees that the state reported to the user contains no stale events (e.g. the problem of hitting a deleted breakpoint is eliminated). Our actual event detection algorithm is:

```
1: Attempt to receive an exception message with a timeout.
If there are any exceptions waiting to be processed
    If this is the first exception since we continued last time,
        suspend the task
    If the exception is a trace trap, process it and come back for more
    If the exception is a breakpoint trap, announce it to the user
```

Continue in this loop until no more exceptions (i.e., goto 1)

- 2: See if any thread stopped with a Unix signal. If so, process it
  - If no Unix signals and we found an exception above, process the last exception found
  - If no exceptions were found, start the entire algorithm over again (i.e., goto 1)

In this description, the term 'process' should be read as 'invoke the higher level debugger code and set the current thread to this thread.'

When the first exception is received, the debugger suspends the entire task. It continues to receive exceptions which have queued up on the exception port and deals with them appropriately, but no further suspension is necessary. It immediately sends a reply message to each thread which hit an exception. The thread is free to continue running upon receipt of the debugger's reply, but since the task is suspended, the thread won't execute until the debugger resumes the task. When control finally passes up to the user, the *current thread* in the user interface will be set to the thread found stopped with a Unix signal if there is one (because this signal must be processed in order to reset *ptrace*), otherwise it will be the last thread for which an exception message was received.

## 6.5 Continuation Logic

A fundamental design decision in our *gdb* makes it impossible to handle more than one breakpoint per continuation without extensive modifications. This restriction is due in part to the *current thread* concept that permits us to utilize existing code; that code can only recover from a breakpoint in the current thread. Our enhanced *gdb* provides the user with workarounds for this problem; multiple breakpointed threads can be single-stepped (clears the breakpoint) or suspended (defers action on the breakpoint until later) before continuing the application. Of course a thread must be the *current thread* before it is single stepped.

## 6.6 Deadlock

The ability to suspend individual threads in an application creates potential deadlock scenarios. Continuing an application that has no runnable threads causes deadlock because the debugger is waiting for the application to do something and the application is waiting for the debugger to resume one or more of its threads. Due to a feature of Unix signals, the only way to recover from such a deadlock is for the user to send SIGKILL to the application<sup>5</sup> (other signals will not work because they do not resume the application). To avoid this type of embarrassing inconvenience, our debugger checks to make sure that at least one thread is runnable before resuming the application; if no thread is runnable, it complains to the user and will not continue the application until at least one thread is resumed. Thread suspension can cause additional deadlocks in the application if all of the running threads need to

---

<sup>5</sup>e.g. kill -9 *pid*

synchronize with threads suspended by the debugger; these deadlocks can be interrupted by typing control-C because there are threads in the application that are not suspended, and therefore able to handle signals.

## 6.7 Delivering Signals

Signal delivery forms an important part of the debugging of complex Unix applications because it allows signal handlers to be tested and monitored under controlled conditions. Most Unix debuggers implement the delivery of arbitrary signals by using the *ptrace* “continue with signal” action. This is not as straightforward for our debugger because the desired thread may not be stopped in *ptrace*, and signals sent to a multithreaded task are handled by the first thread that notices them. We solve this problem by forcing the desired thread into *ptrace* by sending it a signal. To ensure that the thread we are interested in receives the signal, we send a message to the *ux\_handler* specifying that thread and suspend all other threads in the task. Upon continuing the application, that thread will execute, receive the signal from the *ux\_handler*, and stop in *ptrace*. At this point *ptrace* can be used to deliver the desired signal to the thread.

This mechanism can also be used to force a selected thread to initiate a core dump by sending it a fatal signal that causes a core dump. Since resuming the application resumes all of the threads (not just the thread that will generate the core dump), the user must suspend all other threads in order to preserve the application’s memory state. Due to single-threaded limitations in the core file format, core dumps for multithreaded programs currently contain state only for the thread that initiated the core dump. Switching to a new core file format that allowed multiple thread contexts to be dumped would avoid this limitation.

## 7 Debugger Attachment

A debugger that attaches to a running process has been implemented using the techniques described in Section 4 for replacing *ptrace* with other kernel operations. Before using these techniques, the debugger must first obtain task and thread ports for the application. All Mach tasks and threads are represented to the outside world by a port; a user process must have the corresponding port in order to perform operations on a task or thread. Our debugger currently uses the Mach `task_by_unix_pid` kernel call to obtain the task port for a given Unix process id (*pid*); this call checks the Unix uid’s of the application and debugger to prevent a protection violation. The thread ports for threads in the task can be obtained by using the Mach `task_threads` kernel call. Because the *wait* call cannot be used to detect termination of the application in this case (the debugger is not the parent of the debugged application), a notify mechanism based on the death of the task port (caused by application termination) is used to detect application termination.

Attaching to and detaching from an application is transparent because the debugger does not rely on *ptrace* or the STRC “traced” bit to alter application behavior. The debugger

attaches by obtaining the application's task port and suspending the application to examine it. It detaches by resuming the application and deallocating the task port from the debugger. The deallocation happens implicitly if the debugger exits.

Signal behavior is slightly different from the normal case because the application process is not "traced". External event signals no longer cause the application to stop for the debugger, but instead invoke their handlers or carry out the corresponding default actions. Interruption of a runaway process is implemented by deploying a signal handler for SIGINT in the debugger; a control-C invokes this handler to wake up the debugger which then suspends the application for further debugging. Signal delivery to the application is implemented by using the normal Unix kill() mechanism to send signals; this may not override pending signals (unlike the *ptrace* continue with signal mechanism).

## 8 Related Work

[7] describes CodeView 2.2 which provides debugging support for multithreaded programs in OS/2. The OS/2 kernel provides an enhanced *ptrace* interface for interaction between the debugger and the application making possible a clean debugger design and user interface. The major enhancements to *ptrace* are the ability to manipulate any thread in the application (an additional argument specifies the desired thread), and the decoupling of *ptrace* from the semantics of signals [5, 6]. These improvements involve changing the *ptrace* call itself and are made possible in part by the absence of a need to maintain Unix compatibility. Concurrent breakpoints are not mentioned in any of these sources. This might be because OS/2 only runs on uniprocessors where it is possible to avoid concurrent breakpoints via scheduling enhancements (e.g. run debugger immediately in response to the first breakpoint). We did not have this choice because concurrent breakpoints cannot be avoided on multiprocessors that support parallel execution of threads.

The Parasight project at Encore Computer has been following a different approach for debugging multithreaded applications on multiprocessors. [2] Parasight inserts debugging "parasites" into the application to directly monitor its execution state; these "parasites" execute as independent threads on spare processors (not in use by the application) to continually check assertions. Parasight relies heavily on this technique of modifying the application to perform debugging (e.g. breakpoints are implemented by a branch to debugging code inserted into the application by Parasight). In contrast, we have retained the traditional Unix debugger model which avoids modifying the application except for breakpoint insertion.

## 9 Future Enhancements

One major area of work we chose not to attack was a major redesign of *gdb* to incorporate support for multiple threads throughout its code. Such a redesign would include saving state information about every thread so that multiple breakpoints can be handled by a single continue. This would make it possible to support conditional breakpoints and breakpoints



that execute commands and continue without user intervention which cannot be supported in our current design.

In addition, the exception handling processing should be written to take advantage of the detailed information received through the MACH Exception Handling Facility. For example, when the debugger receives a breakpoint trap exception from MACH, it knows whether the exception was caused by a real breakpoint or by a trace trap. Using the *ptrace/wait* combination for finding out an application thread's stop status, all the debugger knows is that the thread stopped on a SIGTRAP which could be either a breakpoint or a trace trap. Our *gdb* is currently implemented to convert the detailed information back to a SIGTRAP so that the original *gdb* code can interpret what to do. Much of the complexity of that code could be simplified by knowing up front what kind of trap occurred.

## 10 Acknowledgements

The Mach kernel features for debugging support (Mach *ptrace* and the exception handling facility) were implemented by David Black at Carnegie Mellon University. Some initial work on adapting *gdb* to multiple threads was done by Karl Hauth at Carnegie Mellon University. The bulk of the work in restructuring *gdb*'s internals to properly deal with multiple threads was done by Deborah Caswell at Hewlett Packard Laboratories. The attachment feature was implemented by Richard Sanzi at Carnegie Mellon University. The authors would like to thank Hewlett Packard for supporting this work. Further detail on the Mach system and the various kernel calls used by our debugger can be found in [1] and [3].

## References

- [1] Accetta, M., R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, Jr. and M. W. Young. "MACH; A New Kernel Foundation for UNIX Development", *Proc. USENIX 1986 Summer Technical Conf. and Exhibition*, 9-13 June 1986, 93-112.
- [2] Aral, Ziya, Ilya Gertner, and Greg Schaffer. "Efficient Debugging Primitives for Multiprocessors", *ASPLOS-III Proceedings (Third International Conference on Architectural Support for Programming Languages and Operating Systems)*. Boston, MA, April 3-6, 1989. pp. 87-95.
- [3] Baron, R. V., D. L. Black, W. Bolosky, J. Chew, D. B. Golub, R. F. Rashid, A. Tevanian, Jr. and M. W. Young. *MACH Kernel Interface Manual*, Dept. of Computer Science, Carnegie-Mellon Univ., 15 Feb. 1988.
- [4] Black, D. L., D. B. Golub, K. Hauth, A. Tevanian, Jr. and R. Sanzi. "The MACH Exception Handling Facility", *Proc. A.C.M. SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, SIGPLAN Notices* 24,1 (May 1988), 45-56.
- [5] Letwin, Gordon. *Inside OS/2* Redmond, Wa.: Microsoft Press, 1988. 289 pages.



- [6] Microsoft Corporation. *Microsoft OS/2 Programmers Reference, Volume 3*. Redmond, Wa.: Microsoft Press, 1988. 430 pages.
- [7] Petzold, Charles. "Techniques for Debugging Multithread OS/2 Programs with Code-View 2.2", *Microsoft Systems Journal*, September 1988, pp. 21-29.
- [8] Stallman, R.M., *GDB Manual; The GNU Source-Level Debugger, 1st ed., GDB Version 2.0*, Free Software Foundation, Jan. 1987.
- [9] Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper and M. W. Young. "MACH Threads and the UNIX Kernel; The Battle for Control", *Proc. USENIX 1987 Summer Technical Conf. and Exhibition*, 8-12 June 1987, 185-197.



**Deborah Caswell**  
HP Labs

Deborah Caswell is a software engineer at Hewlett Packard Laboratories, Palo Alto, CA where she and a colleague have ported the MACH operating system to HP's 68020-based workstation. Her research interest is distributed systems. She has been with Hewlett Packard for 8 years with 3 years in her current position.

Ms. Caswell received her B.A. in Computer Science/Math from Dartmouth College in 1981 and M.S. in Computer Science from Stanford University in 1986. She is a member of the IEEE Computer Society.



**David Black**  
CMU

David L. Black is a graduate student and Ph.D. candidate in the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA where he has worked on the design and implementation of the Mach operating system since 1985. His research interests are in the field of operating systems, with emphasis on multiprocessors and scheduling issues.

Mr. Black received the B.A. and M.A. degrees in Mathematics and the B.S.E. degree in Computer Science and Engineering from the University of Pennsylvania, Philadelphia in 1983, and the M.S. degree in Computer Science from Carnegie Mellon University in 1985. He is a member of ACM, IEEE, and several honorary societies.



# **pdb: A Network Oriented Symbolic Debugger**

*Paul Maybee*

*Solbourne Computer, Inc.*

## **ABSTRACT**

**pdb** is a symbolic debugger designed to support program debugging needs on a network of workstations. **pdb** runs under the X Window System<sup>1</sup>, reads object files and symbol tables generated by existing (Sun<sup>2</sup>) compilers, debugs both local and remote processes, and works on multiple hardware types. **pdb** was constructed using object oriented programming techniques to improve its portability to new machine types, languages, and symbol table formats. **pdb** supports debugging at both the source and assembly level. It is entirely mouse and menu driven, with no command language or syntax outside of that required to formulate expressions in the source language.

## **1. Introduction**

A workstation is a powerful desk side or desk top computer with a bitmapped graphics display and several megabytes of memory. It runs some flavor of UNIX<sup>3</sup>, and may or may not have a local disk. A network of workstations may contain machines of only one type or, more commonly, machines of several different architectures. Often machines other than workstations are on the same network, for example, timesharing machines, scientific computers, or special purpose hardware like hypercube, database and Connection machines. The computers on the network typically share file systems, access to printers, and are usually under a single administrative control. Supporting debugging needs on such a heterogenous network is the goal of the **pdb** project.

The debugger takes advantage of the graphical capabilities of the workstation. The X window system was selected for the interface since it is perceived as the de facto standard workstation windowing system. Unlike other debuggers, **pdb** does not have a command language; there are no commands and no terminal-like windows to type commands into. The interaction with the debugger involves selecting source lines, expressions, and stack frames, popping up menus, and moving around icons.

**pdb** allows debugging of processes anywhere on the network. Debugging a remote processes has always been possible using a remote login to the host and then debugging normally. **pdb** does not require the remote login session. Debugging "anywhere in the network" on a network containing machines of differing architectures requires porting. **pdb** was designed for portability. The different aspects of porting were identified and an attempt was made to isolate the dependent code in specific objects. **pdb** contains objects that isolate the window system, architecture, language, object file format, core file format, and operating system interface. **pdb** has already been successfully ported to different architectures, operating system interfaces and core file formats.

Computers on a network cooperate to perform a variety of tasks. Network cooperation involves communication between processes residing on different machines. Even programs that

---

<sup>1</sup>X Window System is a trademark of MIT

<sup>2</sup>Sun, Sun-3 and Sun-4 are trademarks of Sun Microsystems, Inc.

<sup>3</sup>UNIX is a registered trademark of AT&T

do not explicitly access remote machines may do so implicitly, because the network has made access invisible to the application. **pdb** supports debugging such multi-process programs, even when the machines are of different architectures.

Another program that **pdb** has been designed to help debug is the UNIX kernel. **pdb** supports operating system development by providing debugging assistance from the bootrom level to the application level. **pdb** communicates with system simulators, kernel co-resident debugging agents, and running kernels to support operating system debugging.

## 2. Architecture

**pdb** is composed of two processes. The Display Manager (DM) maintains the user interface and typically (but not necessarily) executes on the user's local host. The Remote Agent (RA) resides at the site of the process being debugged and provides process query and control services to the DM. The DM allows the user to simultaneously debug multiple processes running on different machines of dissimilar architectures. One RA executes for each process being debugged.

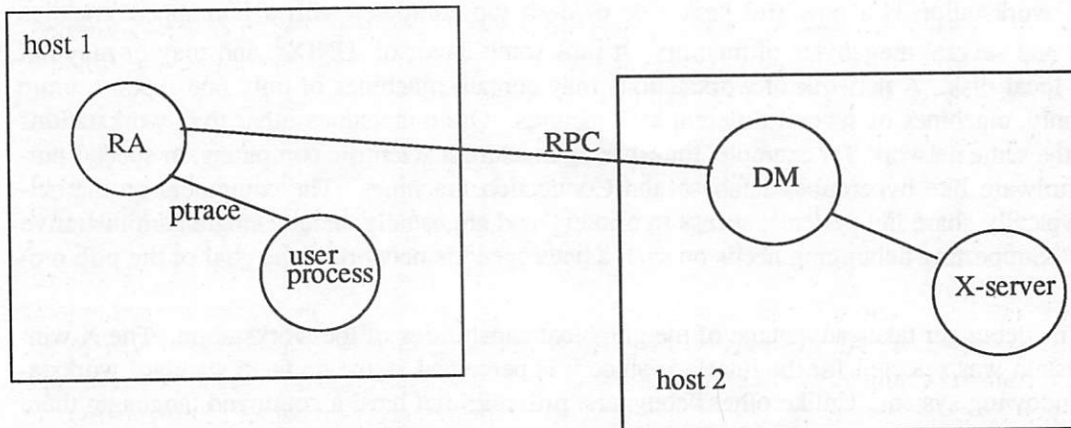


Figure 1: **pdb** architecture

**pdb** is written in C++ (except for C generated by a communication protocol compiler). The object oriented paradigm supported by C++ has proven invaluable in the construction of the debugger (see [Carg87] for a discussion of this topic in reference to the Pi debugger).

The DM is divided into two sets of objects:

- (1) The first to implement the user interface.
- (2) The second to implement the process interface.

The user interface is made up of multiple windows, similar to Pi [Carg85, Carg86], Saber-C [Kauf88] or LightspeedC<sup>4</sup> [THINK88], in the X environment. The process interface provides a process control abstraction for implementing debugging transactions. The user interface translates user input into combinations of calls to the process interface. The DM is independent of the architecture of the host on which the RA is executing. When a DM and RA begin communication the RA informs the DM of any required host specific information (e.g., register names

<sup>4</sup>LightspeedC is a registered trademark of Lightspeed, Inc.

and types). The DM can then construct menus and tables for presenting this information to the user.

The RA process has objects for process control, machine description, code instrumentation, low level operating system interface, core file access, expression evaluation and symbol table maintenance. The process control object implements the remote procedures used by the DM. The machine description object knows the types, sizes, and names of the registers, as well as function calling conventions. The code instrumentation object handles the setting and lifting of break and watch points. The low level interface object knows the operating system specific aspects of process control (e.g., `ptrace` or `/proc`). The core file object knows the file format and address translations necessary for reading a core image. The expression evaluator computes the values of expressions according to the language of the program being debugged. The symbol table object reads program symbol tables and builds a `pdb`-internal representation.

### 3. User Interface

`pdb`'s user interface is composed of several tools. Each tool has its own window and displays information about a single process. When multiple processes are being debugged, multiple sets of windows are present. All windows referring to a particular process can be opened, closed, or deleted together (as well as separately). The set of tools includes a Process Window, Source and Assembly Displays, Expression Evaluator, and a Breakpoint Editor.

The Process Window (Figure 2) is divided into three regions: a control panel, a stack traceback, and a message log. The control panel contains pull down menus for file specification, processes control and for requesting additional tools. The stack traceback is updated every time the debugged process stops. It contains the current process stack or, for a core file, the stack at process termination. In figure 2 the stack traceback window shows four levels of calls in the `csh`, from `main` to `readc`. The solid arrow pointing to `readc` indicates that this is the current frame of focus for the debugger. All the tools will tend to display information for the frame under focus. The focal frame is changed by clicking on the line displaying the frame of interest. The message log window contains any messages that were logged by the debugger on behalf of the process. In figure 2 the debugger has displayed three lines related to reading the symbol table. Only about a third of the symbols were processed on reading the table because `pdb` does a lazy evaluation of the symbol table, i.e., it only reads as much of the table as is necessary to satisfy the current requests. There is a status line at the bottom of each tool. The status line for the Process Window indicates whether the process is executing or stopped, and if stopped, why. For this example the process stopped because it received a SIGSTOP signal.

The Source Display (Figure 3) has a control panel and a window with annotated program source. The control panel contains two buttons and three glyphs representing different types of code instrumentation: breakpoints (stop sign), conditional breakpoints (yield sign), and tracepoints (check mark). The annotations to the source text consist of these instrumentation marks as well as execution focus arrows; a solid arrow for the current point of execution and hollow arrows for call points down the stack. A breakpoint (tracepoint, conditional breakpoint) can be set by pulling a stop sign glyph (check mark, yield sign) from the control panel to a line of source. Process stepping and continuing is controlled by menu/button selections. The first button in the control panel, which in figure 3 is set to perform a continue operation, can be switched to any of the actions contained in the popup menu displayed. The "Control" button contains options for viewing other source text and for specifying directory search paths. Additional actions are available through clicking in the source window. Clicking on a variable causes its



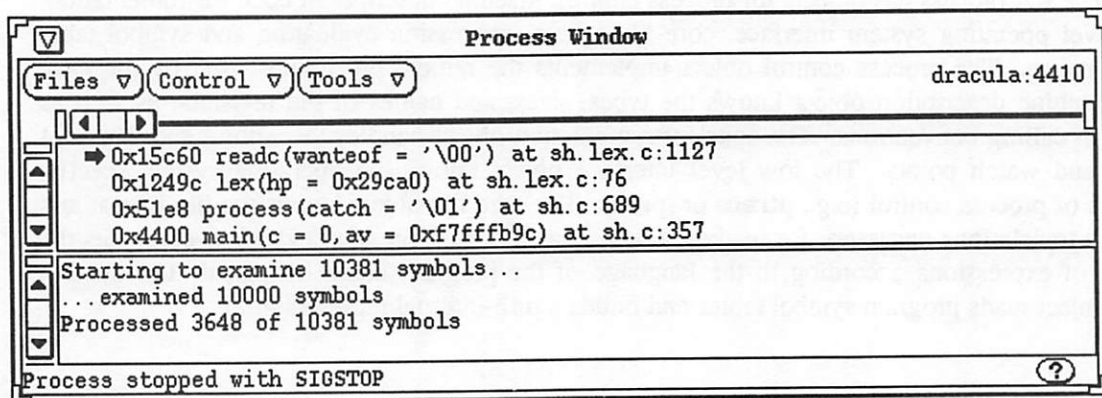


Figure 2: Process Window on csh

value to be printed, clicking on a function name switches the source window to display that function, and dragging the solid arrow to a new source line causes a *once-only* breakpoint to be set on that line and the process to be continued. The Assembly Display (Figure 4) is identical to the Source Display except that annotated disassembly is displayed rather than source.

The Evaluator (Figure 5) displays expressions in some program context. Expressions may be typed in directly to the Evaluator, selected from menus of local, global, and static variables,

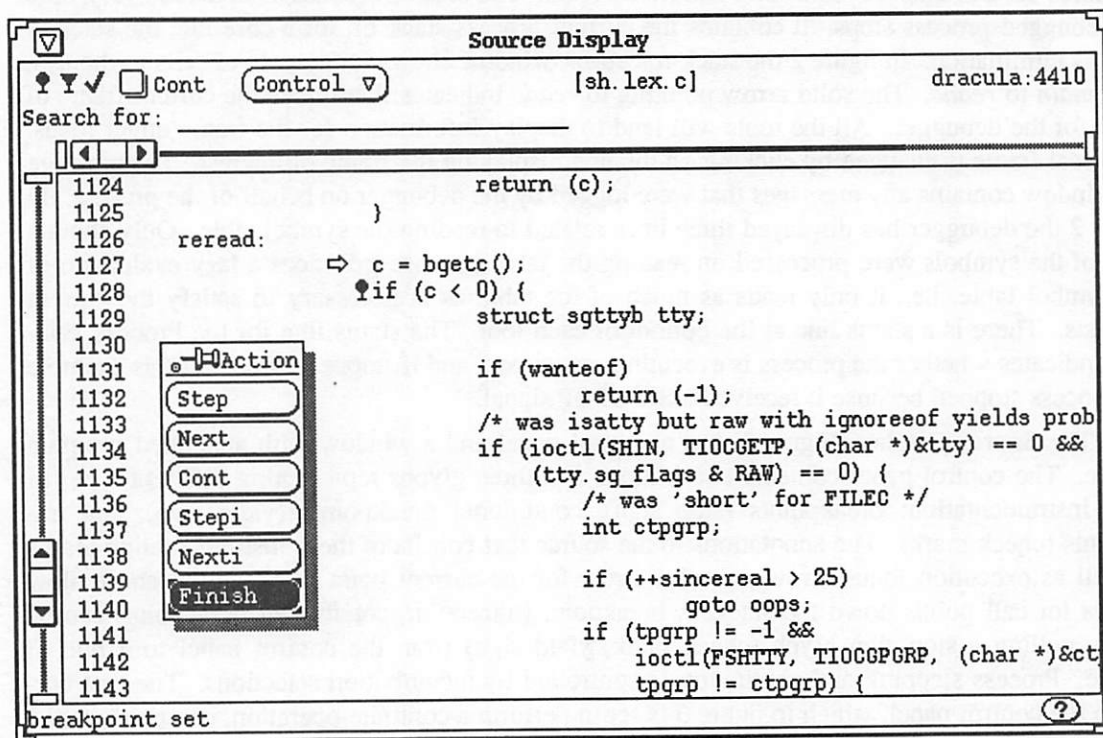


Figure 3: Source Display of csh

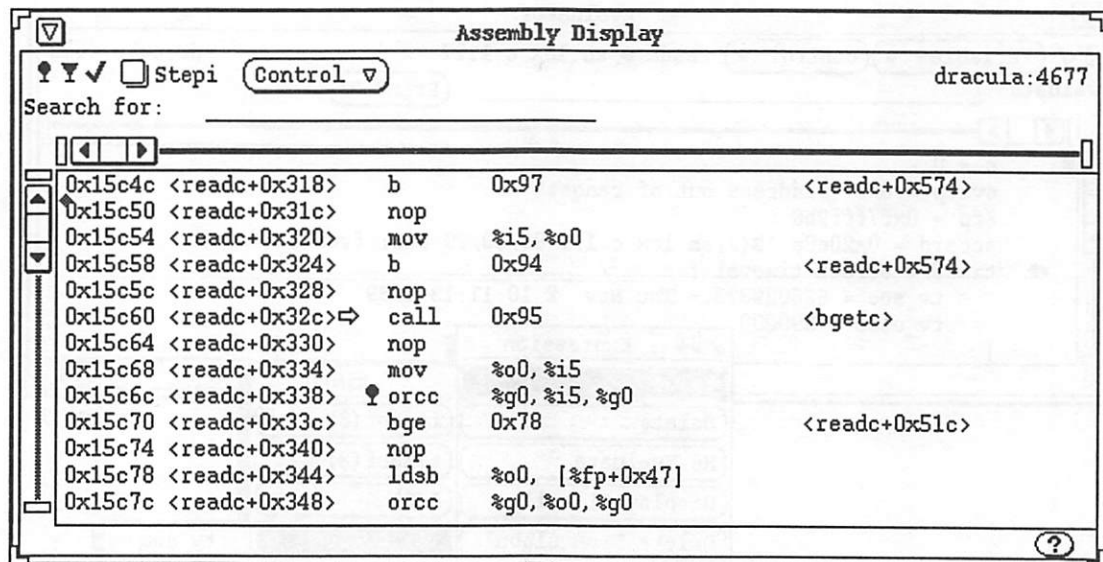


Figure 4: Assembly Display of csh

generated from previous expressions (e.g., by prefacing an existing expression with \*, &, or (type)), or selected from source text. The syntax of expressions is that of the source language (only C currently). Expressions are always evaluated in the current context of the Evaluator. When the Evaluator context changes, in response to program execution or direct user specification, then the expressions for previous contexts are no longer visible. If execution returns to a previous context then the expressions for that context are redisplayed. If no context can be found (e.g., because the process is stopped in a module compiled without symbol table information) then a global context is selected. Expressions are re-evaluated every time that execution of the program stops.

The black arrow is used in the Evaluator to indicate the expression being focused on. When the menu for the tool is popped up the actions on it refer to this expression. Like the Source Display, the Evaluator also contains glyphs in the control panel that may be pulled down. The lock glyph, when placed on an expression, causes that expression to not be automatically re-evaluated each time execution stops. The check mark glyph indicates that the value for the expression should be written to the log every time it is re-evaluated. Any expression that contains an assignment operator, or invokes a program function will automatically be displayed with a lock.

The Breakpoints window (Figure 6) displays a list of all current program instrumentation. Elements of this list can be edited or deleted and new instrumentation can be added. If a breakpoint is deleted then any stop sign glyphs representing it in source and disassembly disappears. Similarly, if a new breakpoint is added then a stop sign will appear if corresponding source (disassembly) is being displayed.

pdb's user interface is built using a C++ X toolkit called OI (object interface) [Aitk89].

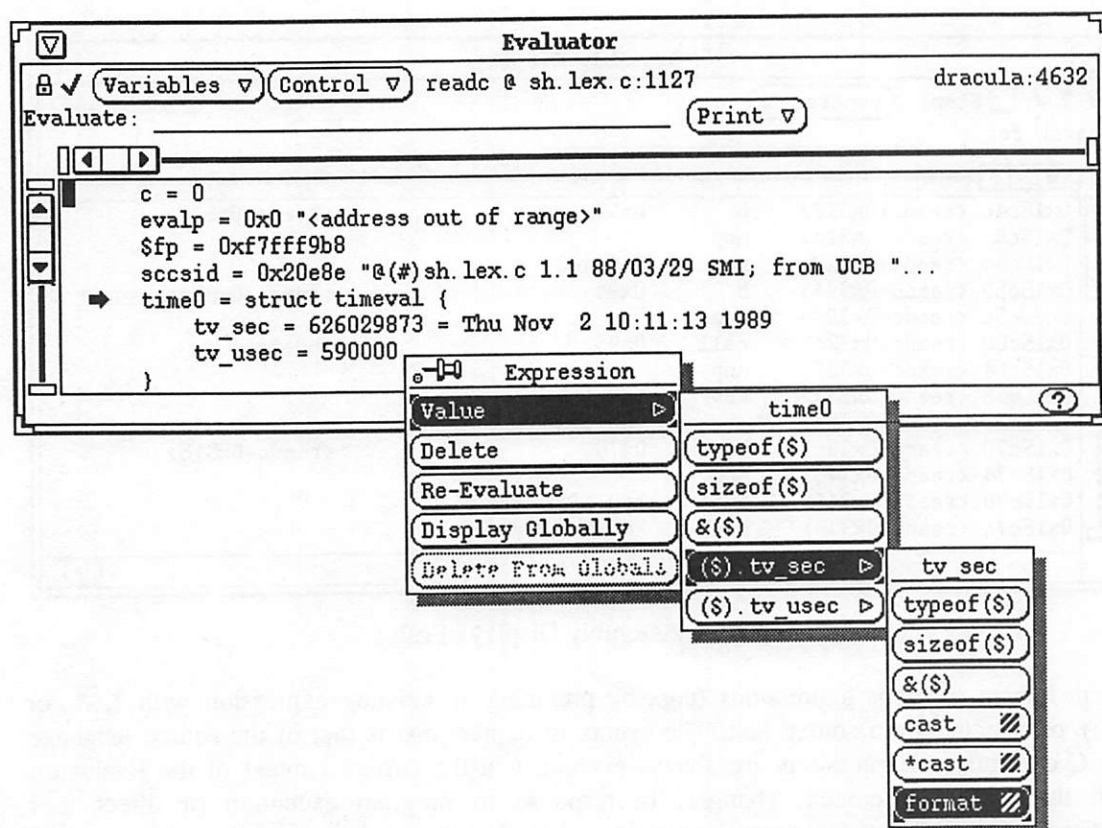


Figure 5: Evaluator displaying csh expressions.

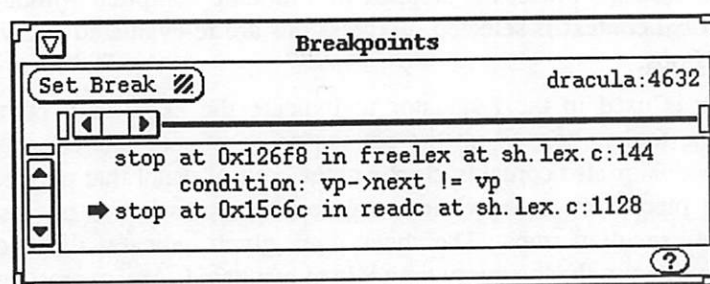


Figure 6: Breakpoints in csh.

The OI implements the look-and-feel of both OpenLook<sup>5</sup> and Motif<sup>6</sup>. The desired look-and-feel is chosen based on the state of a runtime flag. All pictures of **pdb** for this paper were taken using the OpenLook setting.

<sup>5</sup>OpenLook is a trademark of AT&T

<sup>6</sup>Motif is a trademark of the Open Software Foundation

#### 4. Communication

The Display Manager and Remote Agent communicate via remote procedure call (Sun RPC) over TCP/IP connections. The DM causes the RA to be created via the internet daemon and then acts as its client. However the relationship between the two **pdb** processes is not strictly client-server. The RA sometimes needs to inform the DM of the debugged process's activity. When this is necessary the RA makes a call back to the DM bearing the news (e.g., the process has hit a breakpoint, the symbol table is loaded, or the process has executed a traced instruction). These calls are made asynchronously (i.e., no response is expected or waited for) over a separate TCP/IP link.

The set of calls that the DM makes is similar to a standard debugger command set (Table 1). There are six groups of calls: process and object file selection calls, process query and control calls, symbol table query calls, expression evaluation calls, front end configuration calls, and low-level access calls. The process and object file selection calls tell the RA which processes to debug and which files to search for symbol table information. The process query and control calls perform code instrumentation and execution tasks as well as process image querying for stack state and disassembly. The symbol table query calls fetch lists of files, functions, types and variables, and provide a source-to-object location mapping. The expression evaluation calls compute values of expressions in context. The front end configuration calls, at this point, are limited to providing register names and types. Finally, the low-level access calls allow for the transfer of raw (i.e., uninterpreted) data transfer between RA and DM.

The Remote Agent calls (Table 2) indicate the detection of an event by the agent. These are generated in situations for which the DM does not (or cannot) specifically wait. The DM does not wait for the symbol table to load since this may take a long time. The DM cannot block waiting for a process to stop because some processes never stop without input from the user (e.g., sending a signal to the running process). Since these calls are delivered using a reliable communications protocol, and since they do not require a response, the DM does not respond and the RA does not wait for a response. This asynchrony allows for notification of events (especially trace messages) with minimum overhead.

A typical debugging session might proceed as follows. A **pdb** user begins debugging by executing the **pdb** command (the DM process). A process to debug is then specified using the process's host name and pid. If the user wishes to debug a program from the beginning, rather than intercept it while it is running, then the **pause** command can be used. This command starts the process, but suspends it before it executes its first instruction. At this point **pdb** can start to debug. **pdb** makes an ATTACH call to the process's host machine. The RPC number for **pdb** is registered with that host's internet daemon (*inetd*). The *inetd* receives the request and forks a RA process that handles the call. The RA issues an attach request to the process, responds to the Display Manager's ATTACH call, and waits for the process to stop. When the process stops the RA gains control and makes a PROCESSSTOP call to the DM.

The DM then issues a USEOBJECT call containing the name of a file to be read for symbol table information. The RA responds to this call after it has successfully opened the file, but before it has processed the symbols therein. When it has completed it will notify the DM via a SYMTABLEREAD call. For large images symbol table reading takes quite a while and the user often gets anxious about the progress being made. For this reason the RA notifies the DM after every 10000 symbols have been processed with a TRACEMESSAGE that the DM then displays to the user (see Figure 2 above).

Call	Definition
Process and Object Selection	
ATTACH	debug a process
REATTACH	debug a different process
USEOBJECT	use symbols from object file
DETACH	cease debugging a process
Process Query and Control	
TRACEBACK	get a stack trace
SETBREAK	set a breakpoint
DELETEBREAK	delete a breakpoint
SETWATCH	watch a memory location
DELETEWATCH	stop watching
ADVANCE	execute the process
SIGNAL	send a signal to process
HANDLE	handle signals
FETCHASM	get disassembly
Symbol Table Query	
FETCHFILES	get list of files
FETCHFUNCS	get list of functions
FETCHTYPES	get list of types
FETCHVARS	get list of variables
FILLOCATION	get source/object location info
Expression Evaluation	
EVAL	evaluate an expression
EVAL_RES	get an EVAL result
Client Configuration	
REGDESC	get description of registers
Low-Level Access	
FETCHRAWMEM	fetch raw memory
PUTRAWMEM	write raw data to memory
FETCHRAWREG	fetch raw registers
PUTRAWREG	write raw data to registers

Table 1: Display Manager RPC interface.



Call	Definition
PROCESSSTOP	a process has stopped
EXPREVAL	an expression value is available
SYMTABLEREAD	symbol table reading is complete
TRACEMESSAGE	trace message

Table 2: Remote Agent RPC interface.

All the remaining calls, except ADVANCE and EVAL, execute as normal RPC calls. The call is made by the DM, the RA processes the request, returns a result, and waits for the next call. ADVANCE also executes normally, but the process is executing following its completion. Thus the next activity performed by the RA may be either to service another RPC call or to notify the DM that the process has stopped executing again (PROCESSSTOP). The EVAL call also executes as a normal RPC call, most of the time. The exception occurs when the evaluation of the expression causes the process to execute. This happens when the expression contains a user function call. In this case EVAL returns after the process has begun to execute. When the expression evaluation is complete (i.e., after the process has stopped) the RA informs the DM with an EXPREVAL. The RA then uses an EVAL\_RES call to retrieve the result.

The ADVANCE and EVAL calls do not execute synchronously since they rely on user process execution, and thus may take an arbitrarily long time to complete. Because the DM does not wait the user interface is active during process execution, allowing the user to continue to perform actions - stopping execution, for example.

The pdb-internal form of the symbol table is generated lazily from the object file. The symbol table query calls allow the front end to generate menus of file, functions, types and variables. pdb requests the file, function and global variable lists immediately upon starting up, thus the symbol table is initially processed to the extent that these lists can be created. Lists of types and local variables are requested on an as-needed basis, i.e., as the user demands information about the different local contexts of the program. The object file is processed for this information in response to these requests. In a typical debugging session most of the local contexts may never have to be generated [Carg87].

Architectural and implementation specific information is supplied to the DM by the RA via the configuration calls. In the current implementation only a list of register names is supplied. The DM uses the list to generate the Evaluator's registers menu. This group of calls will be expanded (or perhaps a more general specification will be supplied for a single call) to provide more detailed configuration information describing the specific RA.

The low-level access calls read (write) raw data from (to) the process being debugged. The data are transferred by the RA as uninterpreted bytes. These calls are currently unused but may be useful in the context of a raw memory viewer/editor. pdb does not now have such a viewer, but one is planned for the future.

## 5. Operating System Debugging

Software simulators for new hardware are typically ready months in advance of the hardware. Thus, long before new hardware is ready, the process of porting system software can begin. pdb will allow the developers to debug the operating system from the first rom instruction executed on the simulator. Later, when hardware is ready, pdb can help in debugging the

running system through a stand-alone debugging agent interface and it can help in postmortem analysis of failure by reading kernel core dumps. From the first simulation of the bootrom code to modifying the `csb`, `pdb` provides a uniform environment for debugging support.

`pdb` can be used to debug running processes, core files, running kernels, kernel core files, and programs running in simulation - all from the same RA image. The `pdb` user selects the object to debug and the DM then makes an ATTACH call for that object. The RA contains a base class, *process*, and derived classes for each of the possible objects to debug. The derived class for running processes uses `ptrace` for access and control of the UNIX process image. The core file class does not allow any execution or breakpoint operations; access to symbols and stack state is done by reading the core file. A running kernel is debugged over either a SLIP or UDP conversation with the stand-alone debugger executing on a remote machine [Rud89]. Kernel core images are accessed via the kernel virtual memory utilities supplied under Sun OS (*libkvm*). Finally, the derived class for simulator debugging communicates over a TCP/IP connection with a hardware simulator. Both the stand-alone agent and the hardware simulator implement a `ptrace` interface.

The Solbourne operating system supports multiple processors executing a common kernel in which each processor maintains its own stack. `pdb` displays these multiple stacks as preliminary support for debugging shared address space parallel programs. When debugging a multi-stack program, `pdb`'s Process Window displays a "Next Stack" button. Selecting the button will rotate the tool through the stacks. This mechanism is usable for the small numbers involved with a multi-processor kernel, but is not sufficient for debugging programs with large numbers of stacks. Plans for the future include a better method for selecting the stack to display and for displaying multiple stacks simultaneously.

## 6. Current State and Future Directions

`pdb`'s goal was to meet the debugging needs in the network environment. Its current implementation meets these goals in only a limited fashion. Processes anywhere in the network may be debugged (if the host belongs to the class of supported machines), X windows is used to construct the user interface, and a informative debugger interface has been made available. Development is continuing in three important areas: multi-process debugging, multi-language support, and development of a "standard" RPC interface.

The current multi-process support consists of the ability to have windows on multiple processes open at the same time and to debug a restricted class of multi-stack programs. There is no support for debugging forked processes or for controlling or querying multiple processes without switching focus between separate windows. Future versions of `pdb` will include support for debugging forked processes as well as breakpoints that stop more than one process, monitoring of interprocess communication, and collecting and filtering trace output in search of anomalous program behavior.

All users do not program in C. Several languages are typically in use by programmers on any sizable network. More complete C++ debugging support will be added as a first step in making `pdb` a multi-language debugger. Support for other languages (e.g., FORTRAN, Pascal) will then be added as the need arises.

`pdb`'s portability to new operating systems and architectures involves deriving new objects for those portions of the interface that have changed. With the exception of the code for the disassembler, the Sun-3 specific portions of the RA comprise about 700 lines of code, the Sun-4

specific code totals about 1100 lines. Of course both of these implementations use nearly identical operating systems (SunOS 4.0 and its derivatives), so the differences in that aspect are small. The only architectural differences that the RA needed to inform the DM of was the names of registers. Future ports may require expanding the client configuration information to include:

- high or low byte word addressing
- list of signals
- location watching capabilities
- list of unimplemented RPC calls

The last of these indicates that the RA does not have to implement the complete interface. The DM is designed to handle failure of any RPC call. Failures of some calls are catastrophic, however others only result in the inability to generate menus, disassembly, etc. A partial implementation of the RPC interface can still be useful. For instance, the low-level access calls, coupled with ADVANCE and ATTACH, are sufficient for implementing a remote debugger. However the DM process then needs to contain all the architecture-specific code for interpreting the process and object. This functionality was found to be very useful in the early stages of **pdb**'s development.

A completely new RA implementation requires that a small set of RPC calls (currently 25) be constructed. We feel that the interface is already too complex and are working towards a smaller, cleaner interface for basic debugger functionality and a general mechanism for allowing implementations to provide more complex or system dependent features. The overall goal of supporting network debugging cannot be considered complete without the adoption of such a remote agent interface "standard."

## 7. Acknowledgements

This work would not have been possible without the help of Karen Meyer-Arendt, Andrew Gerber and Andrew Rudoff. Karen conspired in the initial design of the project and constructed the symbol table and expression evaluation modules. Andrew Gerber is responsible for the current state of the user interface, including the port to OpenLook. Andrew Rudoff's suggestions, comments, and complaints were a continuing source of inspiration and ideas.

## References

[Aitk89]

Aitken, Gary, *OI: A Model Extendable C++ Toolkit for X Windows*, in submission.

[Carg85]

Cargill, Thomas A., *Implementation of the Blit Debugger*, Software--Practice and Experience, Vol. 15(2), 153-168 (February 1985)

[Carg86]

Cargill, T.A., *The Feel of Pi*, Proceedings Winter 1986 USENIX Technical Conference, Denver, CO (January 15-17, 1986)

[Carg87]

Cargill, T.A., *Pi: A Case Study in Object-Oriented Programming*, C++ Workshop Proceedings, Santa Fe, NM, USENIX Assoc. (Nov 9-10, 1987)

[Kauf88]

Kaufer, Stephen, Russel Lopex, and Sessa Pratap, *Saber-C An Interpreter-based Programming Environment for the C Language*, Proceedings of the Summer 1988 USENIX Conference, San Francisco, CA, (June 20-24, 1988)

[Netw88]

*Network Programming*, Sun Microsystems, Inc., Revision 50 (19 January 1988)

[Rudo89]

Rudoff, Andrew, *Remote Debugging Kernels on Multiprocessor Systems*, in preparation.

[THINK88]

*THINK'S LightspeedC User's Manual*, Symantec Corporation, 1988



Paul Maybee  
Solbourne

Paul Maybee received his B.A. in Mathematics and M.S. in Computer Science from the University of Colorado, Boulder. He is currently an engineer at Solbourne Computer, Inc. In his copious free time he is also working on his Ph.D. at the University of Colorado.



# Some Efficient Architecture Simulation Techniques

*Robert Bedichek*

Department of Computer Science, FR-35

University of Washington

Seattle, Washington 98195

robertb@cs.washington.edu

## ABSTRACT

An efficient simulator for the Motorola 88000 at the ISA (Instruction Set Architecture) level is described. By translating instructions on the fly to a quick-to-execute form we achieve an average ratio of 20 simulator host instructions executed per simulated instruction. Lazy allocation of memory allows large memories to be modelled with low start-up time. We describe our experience using the simulator to develop workstation software. The simulator's speed and extensive I/O device modelling made it possible for us to interactively debug and test a UNIX<sup>®</sup> kernel and diagnostic software well before the hardware was available. Extensions to closely model caches and multiprocessors are sketched.

## 1. Introduction

We present techniques for building a high speed architecture simulator for the Motorola 88000 CPU [1] and CMMU (Cache and Memory Management Unit) [2]. These methods can be used for simulation of other architectures, including CISCs. This work was done while the author was at Tektronix and supported the development of the XD88<sup>®</sup> workstation series.

The concepts described below are implemented in a simulator that runs on 68020-based Tektronix workstations. On a 2.5 MIPS workstation the simulator executes roughly 130,000 88000 instructions per second. The simulator models the 88100 CPU, up to eight 88200 CMMUs, and a number of I/O devices. The simulator has a human interface that gives programmers symbolic debugging facilities. This interface, called the front-end, is derived from dbx. Dbx is a tool normally used for debugging programs that run under UNIX.

The simulator was used to debug pieces of diagnostic code, boot ROMs, a System V<sup>®</sup> UNIX kernel, and other software.

Section 2 explains why we built our simulator. Section 3 gives an overview of the 88000 architecture. Sections 4 through 6 describe our simulator. Some technical advantages of using such a simulator are pointed out in section 7. We explain our not-very-pretty solution to writing this in C in section 8. Our experience with the simulator is discussed in section 9. Section 10 sketches extensions to the simulator.

## 2. Motivation

When CPU architects design a new machine, they typically write an instruction-level simulator to test their ideas. Later, when they are confident of the stability of their design, software engineers are often told to make system software work using the architects' simulator. When the real hardware arrives and is debugged, the software engineers usually switch to using real hardware to test their programs.



The simulators that CPU architects write typically execute thousands of host instructions for every simulated instruction. These simulators are written to test concepts and processor design tradeoffs; flexibility is important and speed is not. Also, they often gather instruction execution time statistics, and this constrains and slows down the simulator. The software engineers, however, would like a simulator that is as fast as possible and is complete enough to run their programs. With a little more work, a simulator can be had that is more complex but is much faster.

We wanted to bring a workstation to market quickly. By building and then using our 88000 simulator, we were able to start debugging software six months earlier than we otherwise would have. The simulator has useful debugging features not available in the actual machine and so is still in use. Some operations, such as downloading text and data, are faster on the simulator than on the real machine.

In the terminology of May [3], ours is a second generation simulator. We translate 88000 instructions to threaded code, while May does flow analysis and generates host instruction sequences with semantics that match the program being simulated. A scheme, similar to ours, for translating target instructions to threaded code is outlined in [4]. Other systems that translate instructions on the fly to a quick-to-execute form include the VAX-8800<sup>®</sup> series of computers [5], the CRISP microprocessor [6], and a Smalltalk 80 interpreter [7].

When Tektronix engineers debug kernels and diagnostic programs on the hardware, they use a cross-debugger that runs on a 68020-based workstation. The workstation communicates with the hardware via an RS-232 serial link. The cross-debugger, simulator, and dbx front-end are all part of a single program (see Figure 1). The cross-debugger and the simulator share the dbx-based user interface. The programmer sees the same human interface for the simulator and for the cross-debugger, allowing engineers to switch easily between simulator and the real hardware.

### 3. Motorola 88000 Architecture at a Glance

Figure 2 shows the system that we simulate. The CPU has separate instruction and data ports. Each CMMU has a 16k byte cache and a 56-entry TLB (Translation Look-aside Buffer). Eight CMMUs are shown, but the simulator and the real system can have one, two, or four CMMUs per CPU port. We call the cache in the CMMU(s) that are connected to the instruction bus the *code cache*; the cache in the CMMU(s) that are connected to the data bus is called the *data cache*. The CMMUs translate a

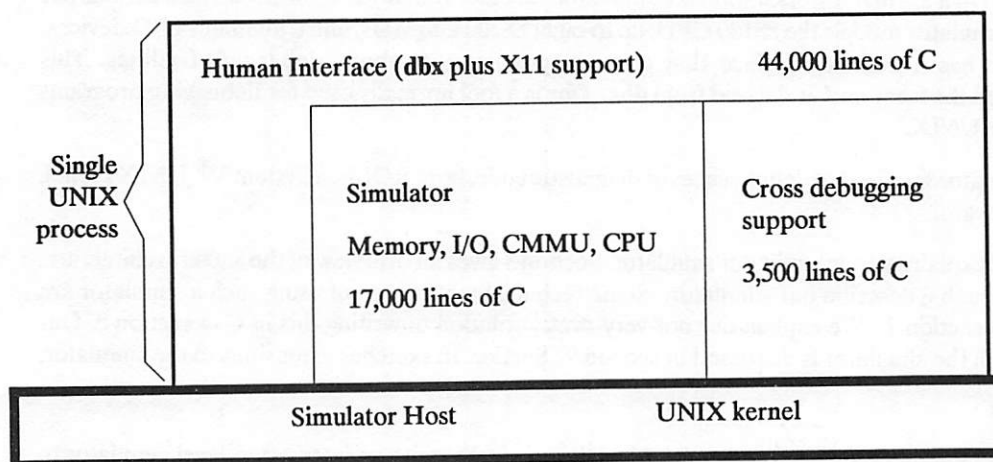


Figure 1: Structure of the simulator

virtual address sent by the CPU to a physical address before accessing the data in cache or, in the case of a cache miss, in memory.

The CPU can concurrently access data memory, fetch instructions, compute floating point results, and execute integer instructions. All of these operations, and external interrupts, can cause exceptions. When an exception occurs the floating point unit is stopped and the contents of several key registers in the instruction fetch unit are frozen in shadow registers. Control is transferred based on the exception type to one of 512 exception handlers. If the processor takes another exception before the shadow registers have been unfrozen, an error exception will occur.

There are two kinds of 88000 branch instructions: non-delayed and delayed. Non-delayed branches execute the branch target after the branch instruction and have an idle cycle. This idle cycle is due to the temporary lack of an instruction to execute while the branch target is fetched, and is called the *branch delay slot*. To take advantage of the branch delay slot the architects included delayed branches. Delayed branch instructions cause another instruction to be executed during the branch delay slot. It gets this instruction from the word following the delayed branch. Any non-branching instruction may be in a branch delay slot.

#### 4. Goals

We wanted a simulator that would:

- Execute the common 88000 instructions as quickly as feasible.
- Make the execution as close to that of the hardware as programmers needed it to be.
- Provide a clean interface to front-ends. This allowed us to switch from an **adb**-based front-end to **dbx** easily. We plan to switch again, this time to a **gdb** front-end.
- Efficiently simulate large memories.
- Have a low start-up time.

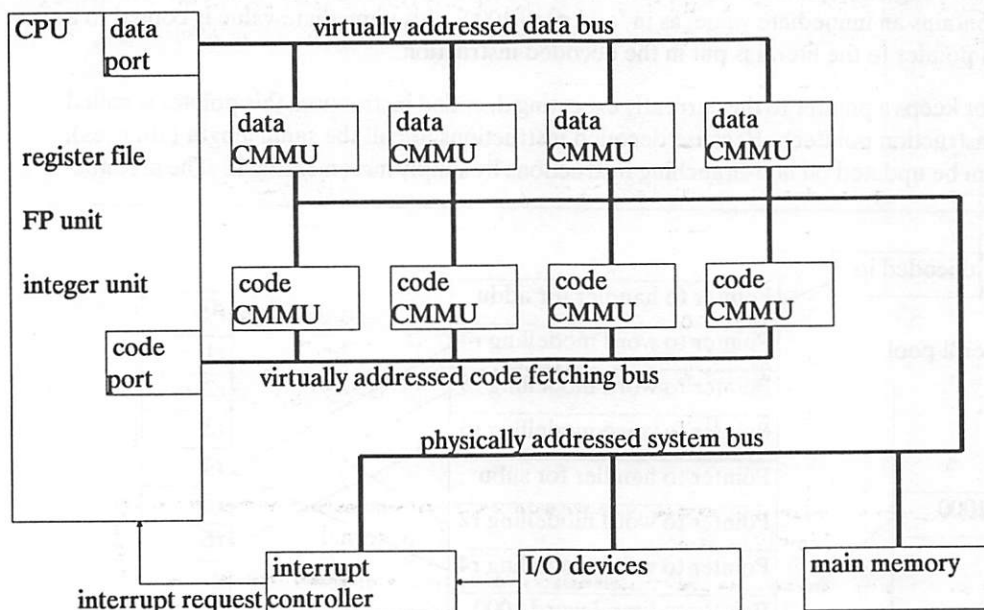


Figure 2: 88000 System Modelled.

- Allow I/O device simulators to be written with little knowledge of the rest of the simulator.

We did not intend to model:

- FP accuracy with respect to the 88000 hardware. We use the host's floating point arithmetic instead of exactly modelling the 88000 floating point unit.
- 88000 instruction timing.
- Little endian mode.
- The instruction and floating point pipelines.
- The exact data cache and PATC (Page Address Translation Cache) contents.
- Transistors, gates, or any other aspect of the physical structure of the hardware.

## 5. Decode Once, Execute Many Times

The simulator does not interpret 88000 instructions directly. Before an instruction executes for the first time, it is translated into a form that can be executed quickly. This translation of an instruction is called a *decoded instruction* and is not visible to the user of the simulator. Decoded instructions are cached in decoded instruction pages. Only instructions that are encountered in the execution of a program are decoded and cached, so there is no start-up penalty for this technique. We call 88000 instructions *raw instructions* to distinguish them from decoded instructions.

Figure 3 shows the decoded form of an unsigned add instruction followed by an unsigned subtract instruction. The add sums the contents of r5 and r6 and stores the result in r4 ("addu r4,r5,r6"). The subtract computes the difference of the contents of r4 and the literal 1000 and puts the result in r2 ("subu r2,r4,1000"). The first field of every decoded instruction is a pointer to the sequence of simulator instructions that execute it. We call this code the instruction's *handler*. Decoded instructions along with their handlers are a form of threaded code [8]. Most handlers are 9 to 35 host instructions long. Decoded instructions that have register operands have pointers to the memory that model these operands. If a raw instruction contains an immediate value, as in "add r9,r9,1000", this immediate value is copied to a literal pool, and a pointer to the literal is put in the decoded instruction.

The simulator keeps a pointer to the currently executing decoded instruction; this pointer is called the *decoded ip* (instruction pointer). Because decoded instructions are all the same length (16 bytes), the decoded ip can be updated on non-branching instructions by simply incrementing it. The architec-

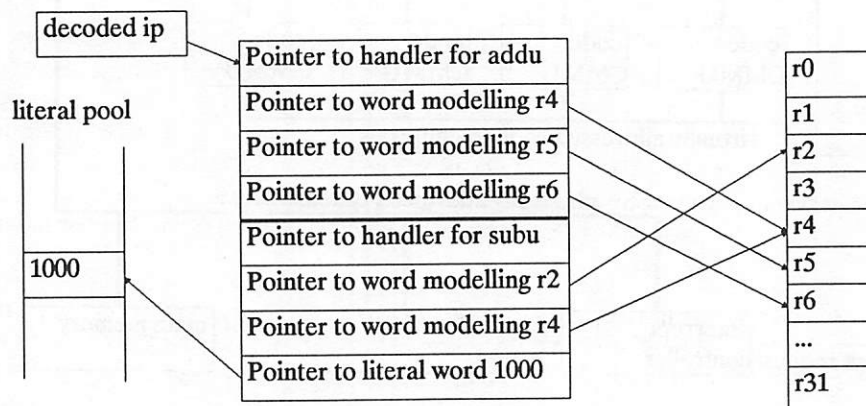


Figure 3: Two consecutive decoded instructions.

tural ip, the one seen by the 88000 programmer, is not kept explicitly. It is computed from the decoded ip when it is needed. This computation takes four 68020 instructions, and only needs to be done for a few of the 88000 instructions. It is a simple computation because all 88000 instructions are the same length (4 bytes) and all decoded instructions are the same length. Not keeping the ip explicitly saves time on all non-branching instructions that would otherwise have to keep the ip up to date. An early version of the simulator kept both the ip and the decoded ip explicitly. Removing the explicit ip made all the non-branching instruction handlers one instruction shorter and freed up a valuable 68020 register for other use. This sped up the simulator by about five percent.

This is the C source of the handler for addu:

```
L(addu);
DST = SRC1 + SRC2;
DISPATCH_NEXT;
```

There are a number of macros used here: L(addu) defines the entry point for the handlers (described in section 8). DST, SRC1, and SRC2 refer to the operands pointed to by fields in the current decoded instruction. DISPATCH\_NEXT increments the decoded ip and branches to the next decoded instruction's handler.

This is the generated 68020 assembly code of addu handler:

```
.globl _sim_addu      ; created by asm inserts in the L macro,
_sim_addu:           ; see section 8. The decoded ip is in register a2
    movl a2@(4),a1    ; a1 := pointer to word modelling the destination register
    movl a2@(12),a0   ; a0 := pointer to word modelling the 2nd source operand register
    movl a0@d0        ; d0 := the second source operand register contents
    movl a2@(8),a0    ; a0 := pointer to word modelling the 1st source operand register
    add  a0@d0        ; d0 := first source operand + second source operand
    movl d0,a1@       ; store the sum in the word modelling the destination register

    movl a3,d3        ; test to see if we are in a branch delay slot.
    bne  L67          ; we usually are not, so we usually do not branch here.
    addl d5,a2        ; advance decoded ip to next instruction (d5 = 16)
    movl a2@a0        ; fetch the address of the next instruction's entry point
    jmp  a0@          ; jump to the next instruction's entry point.
```

Decoded branch instructions whose target is on the same decoded instruction page contain the decoded ip of the target. This makes branches within a page fast. If the target is on a different page, the offset is kept instead, and the decoded ip is calculated each time the branch is executed. This is necessary because the decoded instruction pages correspond to physical 88000 memory pages and branch instructions work with 88000 virtual addresses. A branch instruction may appear at more than one virtual address, or the virtual address of a branch instruction can change after the instruction is decoded. Computing the decoded ip of the target of off-page branches each time the branch is executed preserves the semantics of branches and of the virtual to physical translation mechanism. To reduce the penalty of off-page branches we cache translations of virtual code addresses to decoded instruction pointers. Implementing this cache increased the overall speed of the simulator, when running the UNIX kernel, by ten percent. We invalidate this cache when the code CMMU(s) are told to invalidate their TLBs.

Execution of delayed branches cause the decoded ip to be incremented, as it is for non-branches, and a flag is set if the branch should be taken. Each non-branching instruction handler checks to see if it is in a delay slot. If it is, control passes to the target of the most recent delayed branch instead of to the following instruction. This extra check adds two 68020 instructions to each handler. If the simulator didn't have to implement delayed branches, it would run about 5 percent faster.



When a decoded instruction page is first allocated, and when it is flushed, the decoded instruction slots are filled with an instruction we call the *decode* pseudo instruction. (Actually, just the first 1024 slots are so initialized, see the next paragraph for what happens to the 1025th slot.) When a decode pseudo instruction is executed a raw instruction is translated to a decoded instruction. The raw instruction is fetched from the address in the raw page that corresponds to the position of the decode pseudo instruction in the decoded instruction page. For example, if a decode pseudo instruction in the 10th slot is executed, the 10th word in the corresponding raw page will be translated. The new decoded instruction replaces the decoded pseudo instruction and then this new instruction is executed.

There are 1025 decoded instruction slots in each decoded instruction page. The first 1024 of these hold the decoded form of the 1024 raw instructions that can be in a raw page (a raw page is 4kb). The 1025th has a *requalify* pseudo instruction that causes the decoded ip to be requalified. When a non branching decoded instruction in the 1024th slot is finished executing the decoded ip is incremented and points to the requalify pseudo instruction. Because the flow of control has moved off of the page the decoded ip must be requalified. Having the requalify pseudo instruction in the last slot saves time by making it unnecessary for the handlers to check for the end of page condition.

## 5. Modelling Memory

A large physical memory is simulated with low start-up time by not allocating host memory for simulated physical memory until some simulated instruction or some front-end operation touches it. For example, the front-end causes a raw page to be allocated when the user examines memory on a previously untouched page. The size of active simulated physical memory (i.e., simulated memory that is touched) is limited only by the host's limit on process virtual memory. When the simulator is started, an array of pointers is allocated with one element for each 4k page of physical memory. When an 88000 instruction, I/O device, or front-end operation accesses memory, this table is indexed by the page frame number of the physical address (see Figure 4). If the addressed element is nil, a raw page structure is allocated and a pointer to the new structure is installed in the table. This structure contains a few words of overhead and a 4k byte array that models a page of physical memory.

Like the allocation of memory for simulated physical memory, the allocation of space for decoded instruction pages is lazy. The decoded instruction page corresponding to a simulated physical memory page is allocated when the program being simulated attempts to execute an instruction on that page.

To make memory reference instructions execute quickly, the simulator has a translation look-aside buffer that corresponds in concept to the hardware TLB (Translation Look aside Buffer), but its structure is quite different (see Figure 5). The software TLB is a two level table indexed by the top twenty bits of data virtual addresses, and has different sections for user loads, kernel loads, user stores, and kernel stores. The first level tables are allocated statically and can each point to up to 1024 second level tables. The second level tables are allocated on demand and can each point to 1024 raw pages. If a memory reference instruction handler finds a nil pointer in either a first or second level table, it calls a function to try the operation. This function may fault the transaction, call an I/O simulator, or complete the operation and fill in the software TLB so that subsequent references will operate quickly.

Separating translation buffers into four sections reduces the amount of testing and branching that memory reference instruction handlers must do. The CMMU supports write protecting pages, but the store instruction handlers need not check to see if the page they are about to write to is write-protected because they use a section of the TLB that can only point to raw pages that are not write-protected. The CMMUs also support making a page accessible only in supervisor state. However, the load and store instruction handlers need not check for this, because they use a different part of the TLB depending on which mode the processor is in.

The segment and page tables are examined by the CMMU when the physical address is needed for a logical page that is not in the TLB. This is true on both the simulator and the hardware. Because the



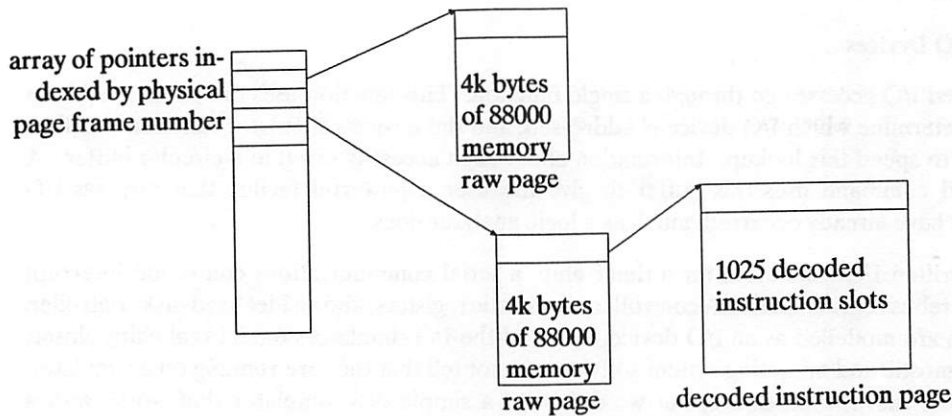


Figure 4: Example of simulation memory after two pages have been touched. One of these pages contains at least one instruction that has executed.

simulator's TLB is different from the real TLB, the segment and page tables may be examined at different times. If a program that manipulates these tables, such as a UNIX kernel, does not flush the TLB at all the right points, errors may result from the presence of a stale translation in the TLB. Because the simulator's TLB is much larger, these errors will be more likely to be noticed on the simulator. This is an advantage of using the simulator.

When a program running on the simulator flushes a code cache, the corresponding decoded instruction pages are cleared. The simulator, therefore, handles self-modifying code just as the real machine does. The difference is that the real machine has from 16K to 64K bytes of instruction cache, depending on the number of CMMUs installed, whereas the simulator has no fixed limit on the number of decoded instruction pages that can accumulate. As with the TLBs, this will make some errors occur more readily on the simulator.

The only aspect of the CMMUs cache and TLB that we modeled closely are the diagnostic ports that allow the cache to be tested. Modelling the caches and TLB more closely would have been of no

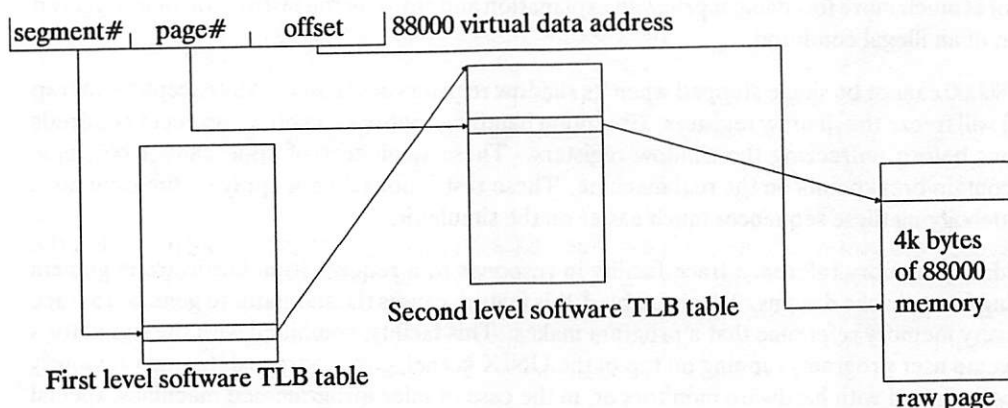


Figure 5: Example of one of the four software TLB data structures that cache the translation of 88000 virtual data address to pointers to pages modelling physical memory.

benefit to us.

## 6. Simulating I/O Devices

All simulated I/O accesses go through a single function. This function uses the physical address passed to it to determine which I/O device is addressed, and the device's simulator function is called. Hashing is used to speed this lookup. Information about each access is saved in a circular buffer. A special front-end command uses this buffer to give the user a powerful facility that displays I/O transactions that have already occurred, much as a logic analyzer does.

We have written I/O simulators for a timer chip, a serial communications controller, interrupt controllers, Futurebus registers, a DMA controller, diagnostic registers, and an idealized disk controller. The boot ROMs are modelled as an I/O device. Most of the I/O simulators model real chips closely enough that diagnostic and operating system software cannot tell that they are running on a simulator. We did not model the disk hardware, but we did write a simple disk simulator that works with a correspondingly simple driver in our UNIX kernel. This simulated disk driver is the only significant piece of our kernel that is special for the simulator, and the kernel switches to it automatically. It does this by looking at the 88100 mask revision register, which for the simulator has a value that we expect no real 88100 to have.

The most complex device simulator we wrote is for the Zilog 8530 SCC (Serial Communications Controller). It is 1351 lines of C, and simulates most of the features of the 8530 when running in asynchronous mode. If we had simulated the synchronous features, which we did not need, it would have been twice as large. Our strategy is to simulate only what we need. When the 88000 program initializes this chip, the SCC simulator opens a host tty line for each of the two ports on the 8530. When the 88000 program changes the bit rate or hardware flow control flags, the SCC simulator makes the necessary host `ioctl` (I/O Control) calls to make the real tty line match the simulated port. This makes it possible to connect terminals to a machine running the simulator and use these terminals as though they were connected to a real 88000 system.

Some features are unavailable when cross-debugging, such as commands to examine write-only registers in I/O devices.

## 7. Tricks That The Hardware Cannot Do

There are some conditions that 88000 code running in supervisor mode should not allow, but can arise through a coding error. If this happens in a kernel that is running on a real machine, the system will stop completely with no indication of error. A logic analyzer is often required to find the problem. The simulator is much more forgiving; it prints an explanation and stops on the instruction that triggered the detection of an illegal condition.

A real 88000 cannot be single stepped when its shadow registers are frozen. Any exception (a trap or interrupt) will freeze the shadow registers. Exception handling routines sometimes execute hundreds of instructions before unfreezing the shadow registers. These sequences of code cannot be single stepped or contain breakpoints on the real machine. These restrictions do not apply on the simulator. This makes debugging these sequences much easier on the simulator.

We added a memory reference trace facility in response to a request from hardware engineers contemplating future cache designs. When enabled, this feature causes the simulator to generate a trace record for every memory reference that a program makes. This facility, combined with the simulator's ability to execute user programs running on top of the UNIX kernel, gives us trace data that previously could only be gathered with hardware monitors or, in the case of microprogrammed machines, special microcode [9].

## 8. Details And Kludges

The simulator is written entirely in C. We used the Green Hills 68020 C compiler. We looked at the generated code and tweaked the C code to get efficient handlers.

There is a kludge that would not be necessary in a language that supported label values. It is the following: all of the handlers are in a single, large procedure. Each handler begins with a macro (e.g., `L(addy)`) that expands to an assembly-language insert that defines a global label (e.g. `asm(" .globl _sim_addu"); asm("_sim_addu");`). Each global label is also declared in C as an external function (e.g. `"extern void sim_addu();"`). The end of each handler jumps indirectly to the handler for the next decoded instruction. In C we write this as an indirect call of the procedure pointer in the first field of the decoded instruction to be executed. This procedure pointer is really a pointer to an instruction sequence. A post-compilation script runs over the assembly code for all the handlers and converts the indirect calls to indirect jump instructions. This scheme is error prone because the compiler is not aware of the true control flow of the procedure and may decide it can reuse registers that it shouldn't. For this reason we have optimization-fooling code in addition to the handlers to force all of the local variables to be live for the whole function. An alternative to this kludge would be to write all the handlers in assembly code. Although it would remove the kludge, it would be no faster, a big job to maintain, and very unportable. Another alternative would be to put all the handlers in a big switch statement and accept about a 40% slowdown due to the switch statement overhead.

## 9. Experience With The Simulator

Kernel engineers extended the simulator to do extra checking to find specific bugs that had been difficult to find when running on the hardware. Diagnostic software engineers wrote a number of device simulators without knowing much about the rest of the simulator. Because the cross-debugger and the simulator use the same dbx-based interface, engineers easily switch back and forth between the simulator and the real hardware. The simulator is still in use, a year after the hardware became available, because it offers features not present in the hardware. The simulator is sometimes used to get a "second opinion" when a software engineer suspects a hardware design flaw. If erroneous behavior occurs on both the real machine and the simulator, it is usually due to a software bug. We expected to find most, but not all, of our 88000 software bugs with the simulator. We could have spent more effort to make a more accurate simulator, but it would be slower and the bugs that we would have found would not have been worth the extra effort.

It took two weeks of intensive coding to get the core of the simulator working well enough to prove the viability of our approach. To date we have spent about six person months on the simulator itself and another six on the front end. A relatively bug-free front end already ported to the 88000 architecture should take a few weeks to interface with the simulator. Our management very much liked the decoupling of hardware and software schedules that was a result of relying on the simulator.

The simulator is made for debugging 88000 kernel and diagnostic code. Finding bugs that required looking at a lot of user-process state is cumbersome. An 88000 UNIX process debugger, such as gdb, would help, but we didn't have one at the time we were first bringing the UNIX kernel up. And such a debugger would be painfully slow in our simulation environment. The simulator is about 20 times slower than the host and about 100 times slower than the real hardware. We can run most UNIX commands on the simulator, including `fsck`, the file system consistency checker. Running a simple command like `date` takes about 20 seconds of elapsed time.

## 10. Contemplated Enhancements

How hard would it be to make our simulator model a multiprocessor and true cache behavior? Is it possible to use these techniques to model machines with variable length instructions? Modest changes are all that is required. Below we sketch the approach we propose for making such changes.

To model multiple processors, one would run some processor's thread for a period of time, call it the *processor interval*, then switch to another processor's thread. Give the user control over the processor interval. The user will set this to a large value for fast simulation and to small value for high accuracy.

Exact code cache behavior can be modelled as follows: allow only those decoded instructions to remain in decoded instruction pages that correspond to raw instructions that could be in a real code CMMU's cache. To do this, keep a table of pointers to the decoded instructions that correspond to the code cache's contents. Use this table to selectively invalidate decoded instructions when a cache line must be reused to make space for a missed line or when the code cache is flushed explicitly. When a miss is taken we currently decode just the current instruction. To model the behavior of the code cache, translate all four instructions in the cache line. With these changes, every decoding of a set of four instructions will correspond exactly to a code cache miss. By looking at our auxiliary data structure we can tell what the exact code cache contents are.

Model the data cache by making the software TLB map to an array modelling the data in the data cache instead of to raw pages. This would require either adding a third level to the software TLB or making the first or second level tables 256 times larger. A software TLB miss would then correspond to a data cache miss. Software TLB entries would then be invalidated when a cache line in a real CMMU would be invalidated.

Machines with variable length instructions complicate the correspondence between raw instructions and decoded instructions. Perhaps the best approach would be to keep the decoded instructions a fixed length, make decoded instruction pages large enough to handle the worst case (i.e., a raw page filled with the shortest raw instructions), and keep a data structure with each decoded instruction page to map decoded instruction pointers to simulated machine virtual addresses.

## 11. Summary

Our technique is to do everything feasible when an instruction is first seen to make subsequent executions fast. This is akin to the compiler-writer's motto "don't do at run time what you can do at compile time." We cache the results of decoding instructions, of translating virtual data addresses to raw host pages, and of translating virtual code addresses to decoded instruction pointers. The simulator is lazy in allocating raw memory, in allocating decoded instruction pages, and in allocating second level software TLB tables. We have a clean interface to the I/O section to make it easy for users to add their own I/O simulators. The interface with the front-end is simple to make it easy to use the good work that other people have put into symbolic debuggers.

There is a need for efficient execution vehicles for the debugging, testing, and measurement of operating system software. Specifications for an architecture usually exist months or years before working hardware is available. In our experience, a high speed simulator provided kernel and diagnostic software engineers with a reliable and inexpensive means of debugging their code six months before the hardware prototypes became available. We had the workstation software ported, and could log in and execute UNIX commands three months before the first hardware prototype was ready.

## Acknowledgements

This work was funded by Tektronix. Andrew Klossner approved the project, helped with the design, extended the simulator, completely rewrote the SCC simulator, and patiently corrected drafts of this paper. Brent Sherwood and Tim Dale wrote a number of the I/O simulators, without which the simulator would be of little use. Robert Henry gave drafts of this paper careful review and pointed out related work. Tony Birnseth and Daniel Klein also gave valuable feedback on drafts of this paper. I thank all of these people both for the technical help that they gave and for their encouragement.



## References

1. MC88100 RISC Microprocessor User's Manual, Motorola Corporation, MC88100UM/AD
2. MC88200 Cache/Memory Management Unit User's Manual, Motorola Corporation, MC88200UM/AD.
3. C. May, "A Fast S/370 Simulator", Proceedings of the ACM SIGPLAN Symposium on Interpreters and Interpretative Techniques", pp. 1-13.
4. Lang, T.G., O'Quin, J.T., and Simpson, R.O., "Threaded Code Interpreter for Object Code", IBM Technical Disclosure Bulletin, 4238-4241 (March 1986).
5. D. W. Clark, "Pipelining and Performance in the VAX-8800 Processor", Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems", October 1987, pp. 173-179.
6. Ditzel, D., and H. McLellan. "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of the 14th Symposium on Computer Architecture, 1987, p. 2.
7. L. Peter Deutsch, "Efficient Implementation of the Smalltalk 80 System", Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages.
8. James R. Bell, "Threaded Code", CACM, June 1973.
9. Anant Agarwal, Richard L. Sites, and Mark Horowitz, "ATUM: A New Technique for Capturing Address Traces Using Microcode", Proceedings of the 13th International Symposium on Computer Architecture, June 1986, pp. 119-127.

XD88 is a trademark of Tektronix, Inc.  
 UNIX and System V are trademarks of AT&T.  
 VAX is a trademark of Digital Equipment Corporation.



Robert C. Bedichek  
*U of Washington*

Robert C. Bedichek received the B.A. and M.Eng. degrees in Computer Science from Cornell University in 1982 and 1983 respectively. He has held several engineering positions and has worked as a consultant for a number of firms. He has done architectural definition, embedded hardware design, and system software development. Since 1988 he has been a graduate student at the University of Washington and he continues to do consulting work. His interests include simulation as it applies to hardware and software engineering, computer architecture, code generation, synthetic image generation, VLSI design, and VLSI design tools.





## Tickerplants on UNIX

*Robert Berkley  
Skip Gilbrech  
Tim Hunt  
Mark Luppi  
Richard Plevin*

Fusion Systems Group  
225 Broadway, 35th floor  
New York, New York 10007

### ABSTRACT

Software tickerplants have long existed in financial environments on mainframes and fault-tolerant machines. Increasingly they are being implemented on UNIX local area networks as well.

Briefly defined, a tickerplant is a software system that receives data from capital markets (stock exchanges, commodity exchanges, money market brokers, etc.) and distributes it to a user community in real time. The community can include portfolio managers, brokers, traders, and financial analysts.

In a sense, one can view tickerplants as descendants of the old tickertape machines, insofar as they are used to place real-time market data on financial users' desks. The resemblance stops there, however. A tickerplant is a composite entity that can be as complex as a commercial RDBMS. The central component of the tickerplant resides on a network server machine, where it fills several roles, including:

- A communications handler that interfaces with one or more high-volume datafeeds from external capital market sources such as stock exchanges.
- A real-time database manager that stores current status for all market instruments, as well as tick-by-tick history for all market transactions. (Even on high-performance machines, no commercial RDBMS can yet sustain market data rates).
- An information bus manager that distributes data to end-user screens and processes that perform real-time analysis of capital markets activity.

Tickerplants are central to the operation of many financial institutions, including fund managers, brokerages, stock exchanges, and commercial, clearing, and investment banks. This paper discusses the tickerplant as a unique product that is appearing in UNIX based financial application environments, after a long history on mainframes and fault-tolerant machines. We focus on the technical constraints and solutions for implementing tickerplants on UNIX local area networks, and highlight the problems in performance, reliability, and network distribution based on our experience in implementing our own tickerplant product.

## 1. Introduction

Real-time market information is the life blood of the capital markets. Traders and financial analysts monitor the flow of market activity, and base their own trading activity on their analysis of this data.

Increasingly, a trader must monitor information from capital markets around the world, including exchanges and brokerage houses. The information must be delivered to the trader's desk with high reliability and minimal delay.

There are a variety of service companies that provide end-to-end delivery of market data from the various capital market sources to the trader's desk. In many cases, however, a financial service company that requires full control over the incoming market data must implement its own software to interface directly with the exchanges. When this software is built to provide market data to a multi-user community, it is commonly called a *tickerplant*.

Tickerplants are designed to capture the incoming stream of market data and route it to user screens and applications. This paper examines the current state of tickerplant technology, and identifies the features that are standard to most tickerplant implementations. Also discussed are reliability and performance issues for tickerplant implementations. The discussion is based on our experience with our own tickerplant product, the Fusion Market Data Server (MDS).

## 2. The Application Environment: Market Data Delivery

*Market data*, in the context of this paper, can be defined as the record of trading activity in the capital markets.\* The trading activity consists mostly of *quotes* (bids and asks of equities, options, bonds, and other market instruments), and *trades* (buys and sells of these instruments).

Market data can potentially come from hundreds of sources. There are, for example, more than a hundred major stock or commodity exchanges internationally. Each of these exchanges keeps track of its trading activity in real time. Since this information is obviously important to the trading community at large, the exchanges provide this data in real time via communication links known as *market data feeds*. These data feeds are implemented using a variety of protocols over both public and private networks (including satellite, microwave, fiber optic, etc.).

In some capital market areas where no central exchanges exist, the major brokers in the market sector provide equivalent data feed outlets to the community at large. Brokerage houses in fixed income markets such as government bonds are the outstanding example of this.

### 2.1. Historical Conditions for Market Data Delivery

Historically, a number of service companies have specialized in providing access to the various market data feeds. These companies---which include Micrognosis, Quotron, Reuters, and many of the fixed-income brokerage houses---manage the entire end-to-end delivery, from the originating exchange or brokerage data feeds, to the terminals (generally leased to the end user) that display the data. Until the advent of digital data feeds (discussed below), the data was usually delivered as a video signal or as full CRT pages.

The outstanding advantage of these services is their packaged delivery of market data. The disadvantage is that the delivered data is not integrated with other vendors' services or systems. This means (among other things) that a separate terminal must be installed for each subscribed service. It is still common to see a trader's desk stacked with several tiers of terminals, most of which are used to monitor various sources of market data and news.

---

\* This is to be distinguished from *market news*, which is the record of economic, political, and corporate events that could conceivably affect the trading activity in the capital markets. Traders obtain real-time market news from separate news services such as Dow Jones and Reuters. Although some tickerplants integrate the handling and dissemination of both market data and news, news services are not discussed further in this paper.

The lack of integration also means that the data is effectively "display-only", i.e. it cannot easily be used as input to real-time analytics, or archived for research purposes in a commercial RDBMS. This is obviously true for data delivered in a video signal. For data delivered in the form of CRT pages, many end users can (and do) write software that extracts the market data from the ASCII stream. This practice is discouraged by vendors of these services, primarily because they charge on a per-terminal basis and are worried about users redistributing the extracted data without paying the additional fee. In some cases, the vendor also sells real-time analytics and historical market data as a separate service; it is therefore not necessarily in the vendor's interest to provide easy access to the embedded data.

## 2.2. Tickerplants

To minimize their dependence on the service companies, and to obtain complete control over the data, users have only one option: developing or purchasing their own software that communicates directly with the exchange or broker data feeds. When this software is an integrated system that includes such features as link recovery, real-time caching and data access, tick-by-tick archiving, and local area network distribution to user workstations, it is considered to be a tickerplant.

Until recently, the cost of developing and maintaining an in-house tickerplant was prohibitively high for most organizations. One reason for this was the price/performance of the available hardware. At intervals of peak market activity, the tickerplant must handle hundreds of market data records ("ticks") per second. The records are frequently compressed, requiring substantial processing power for real-time decompression. Sustaining this transaction rate has historically meant dedicating a mainframe or one of the more powerful super-minicomputers to run the tickerplant software.

A second reason has been the number of interfaces that must be supported. Users at a major financial institution are potentially interested in dozens of market data sources, including exchanges and brokerage houses around the world. Most of these sources have their own variations in protocol, record formats, and link management policy. The in-house tickerplant staff has typically had to integrate all these sources, and specialize in the details of each interface. The tickerplant software must merge these sources into a single stream for delivery to users' desks.

Not surprisingly, many tickerplants have evolved into complex, cumbersome systems, requiring large staffs for development and support. This in turn has made it difficult for many users to have their requirements satisfied. If a department within a large company, for example, requires access to a new market data source, it can wait a long time before the tickerplant staff is able to respond. In frustration, these departments have often gone to outside vendors to get the services they need---defeating the original purpose of building the tickerplant.

## 2.3. Impact of the New Technology

Two recent trends have fundamentally changed the nature of tickerplant development. One development is the rapid emergence of local area networks and UNIX workstations in the financial sector. Server machines for these networks now have the capacity to handle the incoming market data feed traffic at a very attractive price/performance ratio.

A second development is the advent of record-oriented, or *digital data feeds*. These are communications links that integrate the data from multiple capital market sources on a single wire. Rather than a video signal or ASCII pages, they provide the market data as a stream of records in a single generic format. The records are suitable for display, storing in a database, or direct input to real-time analytics. The digital data feeds give the user full control over the data, while off-loading the responsibility of interfacing with the exchanges.

These new digital data feeds are being offered by Reuters and Telekurs, among others. Reuters' Marketfeed 2000, for example, provides reliable delivery of tick-by-tick data from more than a hundred international exchanges on a single 56kb wire. Other vendors that presently offer leased-terminal services are feeling increased pressure to develop their own digital data feed



products.

As a result of the new technology, even small companies and branch offices can now afford an in-house tickerplant. The hardware costs are low, since most high end Unix workstations or servers have the capacity to run an entire tickerplant. Equally important, the new digital data feeds have reduced the cost of developing and maintaining the tickerplant communication software by at least an order of magnitude.

The move toward the new technology has been given impetus by the increasing internationalization of the capital markets, and the explosive growth in financial services. Many tickerplants, for example, could not sustain the huge transaction rates during the October 1987 market crash. This has led to a widespread interest in rewriting the tickerplants that currently run on mainframes and fault-tolerant machines. In turn this has opened market opportunities for the providers of tickerplants that run on UNIX local area networks.

### 3. Tickerplant Architecture

Most tickerplants provide a standard set of features. A tickerplant's central component is a memory-resident database, containing the latest status and information for all tracked instruments. The database must reside in memory to support the high update rate. (There are a few tickerplants that use various caching schemes to avoid keeping the entire database always in memory. Even for these systems, however, there must be sufficient memory to contain the greater part of the database during periods of peak market activity).

A tickerplant will usually include an archive file for the input tick stream, and a tick-by-tick history file sorted by instrument ID. This data can be queried in real-time. Application programs can also register their interest in receiving tick-by-tick activity for particular market instruments (e.g., DEC or GM) by posting their requests to one or more "watchlists." As each tick is received, the tickerplant consults the active watchlists, and routes the data to the interested applications.

There is usually a communication handler process for each incoming data feed. Additionally, there are facilities for data recovery (in the event of machine or data feed failure), and for distribution of the received data to the user community.

The Fusion Market Data Server (MDS) includes the core set of components required for this type of product. These components include the following:

- **Communications handler.** A communications handler process is implemented for each digital data feed that talks to the tickerplant. The communications handler for Reuters' Marketfeed 2000, for example, manages the Marketfeed 2000 LAPB interface, does Huffman decompression of the received packets, and assembles them into messages. It places the messages on a shared-memory message queue to the database manager.

A desirable objective is to isolate (as far as possible) all the feed-specific translation within the communications handler. The handler should produce messages in a generic format to be processed by the rest of the tickerplant. In this way, the rest of the tickerplant is "feed-independent"---it can provide common processing for data from multiple heterogeneous sources.

- **Real-time database manager:** This process receives the tick stream from the communication handlers, and stores current records for all market instruments in a customized memory-resident database. It also maintains an intraday tick-by-tick history for all market transactions.

(It should be noted here that there have been a number of attempts to implement a tickerplant as an extension to a commercial relational database management system (RDBMS), and to enter the market data stream directly into the RDBMS. As far as we know, none of these attempts have been successful. No commercial RDBMS can presently sustain peak rates of market data in real time).



- **Watchlist distribution manager.** This process receives the tick stream from the database manager, and routes it to workstation applications that are "watching" the data. A user application can register its interest in a set of market instruments by sending a *watchlist request* to the distribution manager. The request includes the list of instruments to be monitored (DEC, GM, etc.). Whenever the information for one of these instruments changes, the new information is distributed to all registered applications, including user screens and processes doing real-time analysis.

The interaction among tickerplant processes must be carefully controlled to ensure reliable delivery, and to prevent bottlenecks from developing that could cause the loss of critical market data. The following sections discuss the performance and reliability issues for UNIX based software tickerplants.

#### 4. Realtime Performance

Realtime performance is central to the design of a tickerplant. There are two objectives: (1) minimizing the latency of each tick (the time from its appearance on the data feed to its display on the user screen); and (2) preventing bottlenecks that could cause the loss of incoming ticks. To achieve these objectives, tickerplants must meet the following requirements:

- *The incoming market data stream must be served at a very high priority.* Ideally, not a single tick should be lost, even at peak market rates of several hundred ticks per second. This means that the communications handler process for the tickerplant should seldom or never block on anything but a *read* of the communications line.  
The MDS communication handler process, for example, is designed specifically to capture each tick as soon as it arrives. Its role is limited to reading the incoming data stream, translating each datafeed record into a generic feed-independent format, and adding the record to an outbound message queue. The handler can block only on *reads* of the datafeed or on a full message queue. Great care has been taken to minimize the second case. The size of the message queue is tuneable, so that it can be made arbitrarily large depending on available memory. The process on the receiving end of the queue---the tickerplant real-time database manager---is designed so that it gives highest priority to emptying the queue.
- *The tickerplant must be implemented to conserve CPU resources and avoid system bottlenecks.* Measures we have taken to achieve this include:
  - i. Using low-overhead shared memory queues for routing the flow of tick messages among the tickerplant processes. Except for network distribution, high-overhead IPC mechanisms such as sockets are avoided.
  - ii. Using customized assembly-language semaphores for interlocking among processes. These semaphores are two orders of magnitude faster than the standard UNIX System V semaphores.
  - iii. Keeping archive files and tickstream histories on raw disk partitions, rather than in the UNIX file stream. This gives a stronger guarantee of real-time performance, and ensures the data is on disk in the event of a system failure.
  - iv. Making great efforts to profile our code and eliminate "hot spots." This is especially important for code that handles the tick messages, since it is potentially executed several hundred times a second. As part of this optimization, the use of system calls is minimized whenever possible.
- *The important tickerplant processes, including the communications handler and the real-time database manager, must seldom or never be preempted because of memory or CPU contention.* Paging traffic during intervals of peak market activity will quickly bring a tickerplant to its knees. The server machine should therefore be equipped with enough physical memory to hold the working sets of all active processes. Additional measures to be taken are: (i) locking the tickerplant processes into memory (where this facility is available); and (ii) on multiprocessor systems,

locking the tickerplant processes onto particular CPUs.

Multiprocessor machines such as the Pyramid machine are in fact well-suited to the role of tickerplant servers. If CPU locking is available, other applications can be run concurrently on the server without affecting the tickerplant's handling of the market data stream. This can be a critical advantage since these applications can potentially filter or archive much of the market data locally, without spilling each tick onto the network. In some application situations, this can lead to great savings in network traffic.

A single-processor machine such as a Sun 4 server has more than enough capacity to run the Fusion MDS. In this case, however, tight control over the whole environment is required. Since most present-day UNIX systems do not permit an absolute, non-degradable scheduling priority to be assigned to an individual UNIX process, other applications cannot be run concurrently on the server machine. Real-time response can be achieved only if the server is entirely dedicated to the tickerplant.

#### 4.1. Realtime Performance and Reliable Broadcast

The biggest performance issue for tickerplants is network distribution. Tickerplants must deliver large volumes of market data with high reliability and minimal latency to multiple workstations on a local area network. Either virtual circuits or broadcast messages can be used for this delivery.

Reliable delivery usually implies the use of point-to-point virtual circuits. The problem with virtual circuits is that they consume lots of resources. Delivering a single IBM "tick" to a hundred workstations requires a hundred message deliveries. For large networks, this kind of traffic has a major impact on server CPU and network resources.

Even for large networks, however, virtual circuits are still feasible with appropriate hardware and network topology. A possible solution is to use a powerful multiprocessor as the central server of multiple independent Ethernet segments. Each segment contains a subset of the user community. The multiprocessor provides the requisite CPU resources, and the partitioning of the user community relieves the burden on any particular network segment.

Another solution is to use broadcast distribution. A broadcast protocol minimizes the demands on server CPU and network bandwidth, as well as the latency for delivery of a common record image to multiple sites.\*

Although broadcast distribution is normally not reliable, a number of companies (mostly in the financial sector) have implemented reliable broadcast protocols on top of the UDP/IP layers. Most of these implementations use a low-overhead "negative acknowledgment" approach, whereby the workstation is responsible for detecting gaps in the sequence numbers of broadcast messages, and informing the server about the failed deliveries.

Since these implementations are usually proprietary, they are unreviewed by outside organizations. For this reason, we have implemented the initial version of the Fusion MDS using virtual circuits. A version of MDS exists on the Pyramid multiprocessor, to meet the requirements of larger networks.

We are currently examining the available proprietary protocols for eventual use in the MDS. What we would really like to see, however, is more work by standards committees on public-domain protocols for reliable broadcast.

---

\* It should be noted that broadcast distribution can potentially degrade workstation (as opposed to server) performance, since each workstation is forced to examine every broadcast message whether it requested it or not. When there are hundreds of messages a second, this can consume a noticeable amount of workstation CPU. "Multicast" distribution---broadcasting a message only to the subset of interested workstations---is the preferable solution).

## 5. Network Distribution

An overriding need for tickerplant environments, and the subject of much current development, are high-level "software buses" built on top of the network distribution mechanism. These buses provide logical ports for passing generic "objects" such as market data records or analytic results among applications. Diverse applications can therefore interface with the tickerplant at a high level, taking what they need from the market data stream and folding their own analytic results back into it.

These are familiar concepts to developers of object-oriented software. Spreadsheets and other familiar products are also increasingly moving in this direction, with the ability (for example) to provide "hot links" between spreadsheet cells and external data sources.

The difference for tickerplants is the enormous number of real-time transactions that must flow through the software bus. The tickerplant must not only deliver real-time market data, but increasingly, real-time analytic results based on the market data.

The performance constraints require customized solutions (at least for now). The initial version of the Fusion MDS allows applications to register "watchlists" of objects that they wish to receive. The objects can be market data records, or analytic results from other applications. In other words, an application can write its own records back into the data stream, for forwarding to other applications via the MDS network distribution mechanisms. (An example of this type of analytical result would be a locally computed stock market statistic or industry average. Many trader environments maintain their own private statistics in addition to public statistics such as the Dow Jones Industrials Average).

These applications, or "analytical agents," can be attached to the software bus either as autonomous UNIX processes, or as callback routines within the database manager itself. The latter facility is required because, in many cases, the computed result must immediately follow the received data. In this case, when a new market data record arrives for (say) IBM, the database manager invokes callback routines that have registered their interest in IBM. These compute results based on the received information, which are inserted into the data stream immediately after the tick record.

## 6. Query Servers

An important tickerplant component is the query handler that processes user queries of the real-time database. The Fusion MDS supports queries of current status and tick-by-tick intraday history for each market instrument in the database. A user can, for example, ask for IBM's current price, or for IBM's tick-by-tick activity for the past two hours.

We have used Sybase, Inc.'s Open Server technology to implement the query handler. The Open Server library enables network server processes to provide an SQL-like interface to application programs. An application can use DB-Library (Sybase's C language library for interfacing to the SQL Server) to set up and transmit an SQL-like request to an Open Server process. The application receives the returned data in tabular form, a row at a time, just as if querying a table in the Sybase RDBMS. Front-end tools for interacting with the Sybase RDBMS (such as Sybase's `isql` and Fusion System's `fsql`) can also communicate directly with the Open Server query handler.

This approach gives us the means to support complex queries that span both the tickerplant database and the Sybase RDBMS. An `fsql` user could request the current price and year-to-date price history of IBM stocks. In this case, the query handler would parse the query, fetch the current price from the real-time database, and fetch the price history from tables in the Sybase RDBMS. The data would be assembled from the various sources and returned in tabular form as a single result.

We foresee an increasing need for network servers that provide users with an integrated query interface to access data from heterogeneous sources. Using Sybase's Open Server as a platform, we intend to evolve the tickerplant query handler into a generic query server that can access a variety of data channels, including gateways to mainframe databases.



## 7. Reliability

Tickerplants must be "highly available." Outages can be tolerated, but in many financial environments they must not interrupt operations for more than a few minutes. Use of fault-tolerant servers will solve many of these problems. (Fault-tolerant companies such as Stratus and Tandem are said to be moving towards UNIX). These machines are expensive, however, and are not immune to failures from transient software bugs, system administrator errors, or network jamming.

We have addressed the need for high availability by providing a "fast restart" capability in the initial release of the Fusion MDS. In this scheme, the real-time database is periodically written to a raw disk partition. Since this can be done entirely through the DMA controller, it has minimal impact on system resources. During recovery after a server failure, the stored database image can be quickly reloaded. Transactions occurring since the last backup are replayed from the archive log. This avoids a lengthy regeneration of system state by a playback of the transaction log from the start of day.

A subsequent release of the Fusion MDS will use a redundant-server scheme to ensure high availability. In this approach, two instances of the tickerplant run on separate server machines, interfacing with separate lines from the same data feed service. Applications establish connections simultaneously to both tickerplant servers. One server is the primary server, and the other server is the alternate. The MDS application interface library hides the details of establishing and maintaining this dual connection. Either server can be assigned a primary or alternate role for a particular application (i.e., the alternate server is not a passive standby). We will use a load-balancing scheme to allocate the two servers among applications.

All watchlist requests are sent to both servers. In the event of primary server failure, the application automatically switches to the alternate server. The tickerplant on this machine contains the complete state information for the application, including all active watchlists. The alternate server assumes responsibility for transmitting the watched data to the application.

This type of scheme is relatively straightforward to implement. The only major problem is ensuring "glitchless" switchover---i.e., no lost ticks. This requires the alternate server to buffer the outgoing tick stream, so that market data records lost during the switchover can be retrieved from the buffer and re-transmitted to the application.

To indicate the starting point for re-transmission, the application sends the sequence number of the last received message to the alternate server. This implies that the primary and alternate servers must synchronize the sequence numbers of all outbound ticks.

## 8. Conclusion

This paper has described the major features of a typical tickerplant, as well as the performance and reliability requirements that must be reconciled. Although building a tickerplant is still a complex balancing act, the implementor's task has been greatly facilitated by the recent availability of integrated digital data feeds and inexpensive UNIX servers.

There are still many areas that could use improvement. We would like to see a commercially available network infrastructure for building redundant network servers that meets our performance and reliability needs. Everyone, including ourselves, ends up building customized software to support redundant servers. We would also like to see a public domain reliable broadcast protocol suitable for use in financial applications.

**Robert Berkley**  
*Fusion Group*

Robert Berkley, co-founder of the Fusion Group, has extensive experience in the management of large software development projects, and in workstation networks and database management systems for the financial sector. Before founding the Fusion Group in 1988, he consulted to several of the major investment banks in New York City.



**Skip Gilbrech**  
*Fusion Group*

Skip Gilbrech holds a bachelor's degree in Anthropology (UCLA) and a master's degree in Psychology (Antioch College). Before joining the Fusion Group in early 1989, he was a Senior Systems Consultant in PaineWebber's Capital Markets Division, where he was instrumental in the development of the IHAS (Intelligent Hedging Advisory) System, which provided traders with hedging information and guidance in real-time. Prior to that he was at Goldman Sachs, where he was responsible for implementing an IBM 3295 Plasma Display multiple session workstation system for OTC traders.

**Timothy Hunt**  
*Fusion Group*

Timothy Hunt is a principal in the Fusion Group. Previously he worked for a major brokerage in New York City. He has extensive experience in the use of workstations and Unix for the investment banking industry. Before his move to New York, he worked for a major computer graphics company in California for eight years.

**Mark Luppi**  
*Fusion Group*

Mark Luppi holds a bachelor's degree in French from Brown University and a master's degree in Computer Science from the University of Rhode Island. Before joining Fusion, he helped build tickerplants and reliable workstation networks at Morgan Stanley. Prior to that, he was a developer in the UNIX Development Laboratory at AT&T Bell Laboratories, where he worked on System V memory management and the Remote File System (RFS).



**Richard Plevin**  
*Fusion Group*

Richard Plevin, co-founder of the Fusion Group, holds bachelor's and master's degrees in computer science from SUNY Albany and Yale University, respectively. Prior to founding the Fusion Group in 1988, he consulted to numerous firms in the New York financial community, developing various expert systems for securities trading and insurance.

# GENESIS & XODUS

## General Purpose Neural Network Simulation Tool

*Matt Wilson (wilson@smaug.cns.caltech.edu)*  
*John Uhley (uhley@smaug.cns.caltech.edu)*  
*Upinder Bhalla (bhalla@smaug.cns.caltech.edu)*  
*David Bilitch (dhb@bek-mc.cns.caltech.edu)*  
*Mark Nelson (nelson@smaug.cns.caltech.edu)*  
*James Bower (jbower@caltech.bitnet)*

Division of Computation and Neural Systems  
California Institute of Technology  
Mailstop 216-76  
Pasadena, California 91125

### ABSTRACT

The GENESIS package was created to provide a platform for the rapid development of new neural network simulations with graphical user interfaces. Although targeted for the neural network community, the interpretive front-end and object oriented structures present in the GENESIS platform illustrate a powerful technique which may be useful in the development and implementation of other interactive graphical applications.

### 1.0 INTRODUCTION

Over the last several years there has been a significant increase in the use of computer simulations in the study of the structure and function of biological and computational neural networks. We have built GENESIS (GEneral Network SIMulation System) and its graphical interface, XODUS (X-based Output and Display Utility for Simulators) to provide a standardized and flexible means of constructing neural network simulations and their graphical interfaces. GENESIS was created to provide neural network researchers with an interactive object oriented development tool similar to those available to the CAD engineering community. The package was designed with both ease of use and simplicity of functional enhancement in mind. Graphically, the user constructs interfaces like the one shown in figure A.

Although GENESIS is intended for neural network modeling, the underlying design philosophy of this interactive development tool can also be applied to assist programmers in the creation of many other kinds of software packages.

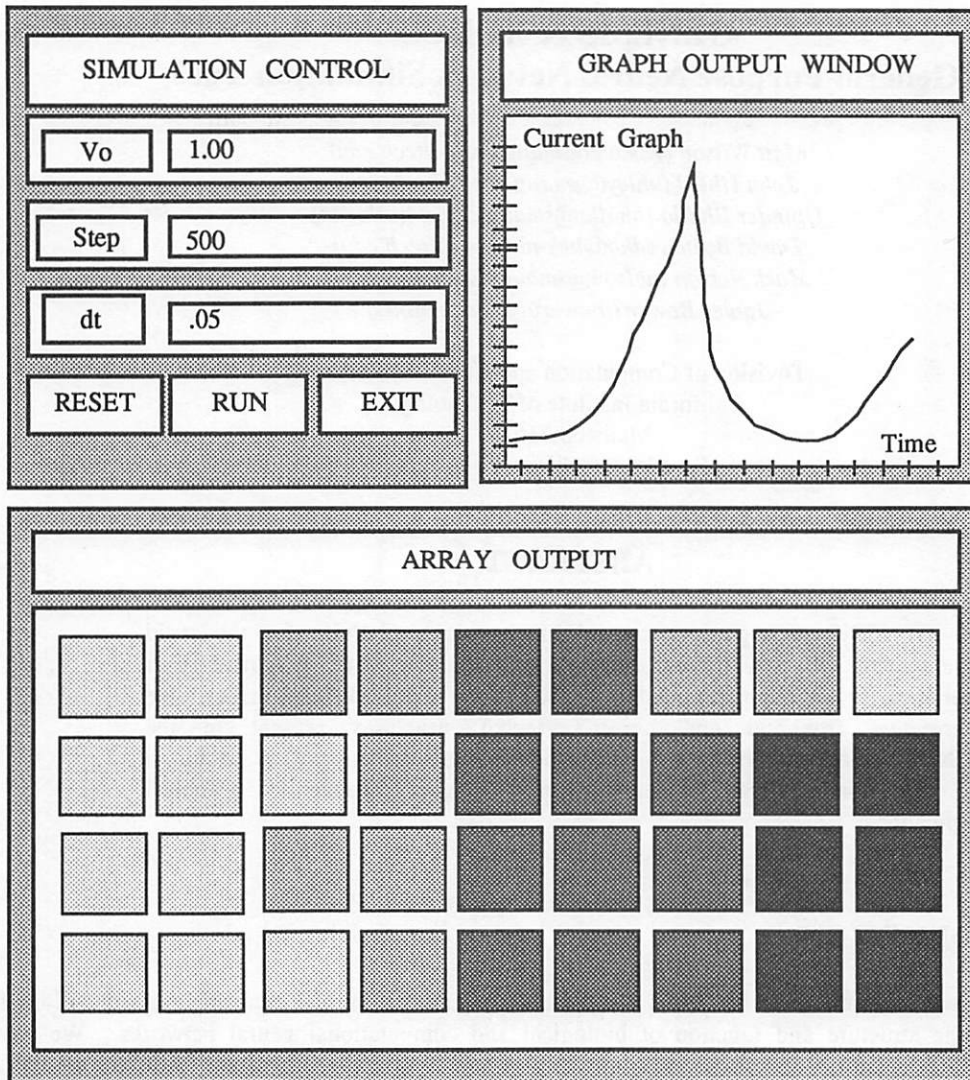
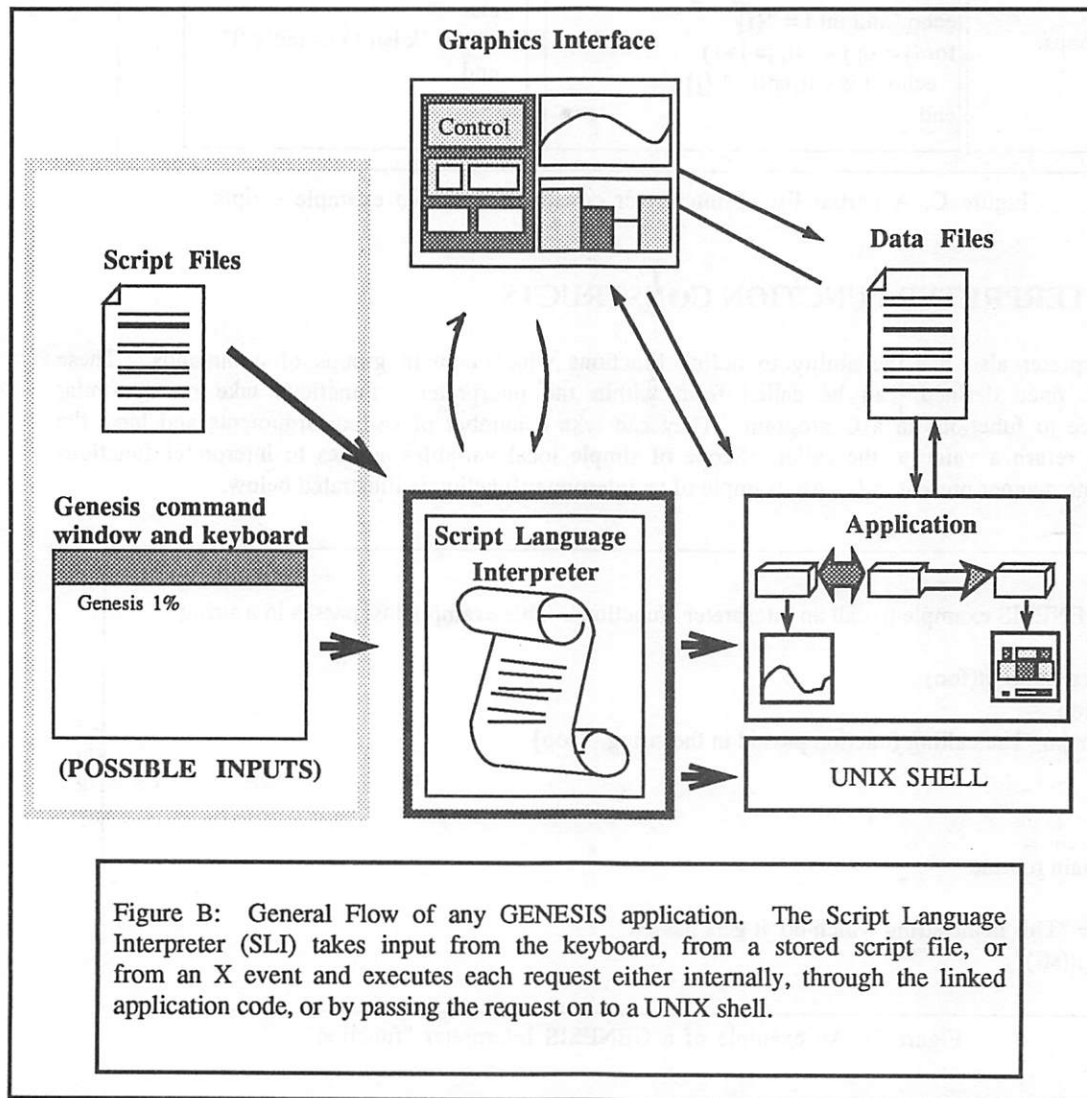


Figure A: An example of a graphical interface in GENESIS is shown above. The control panel allows a user to input initial condition parameters and to define the time step used in the simulation process. Several buttons exist to reset, execute, or exit the simulation. Output, in the form of a graph plotting current vs. time, is generated as the simulation progresses. A second output display uses shaded squares to graphically display an array of integers containing a known range of values.

## 2.0 THE GENESIS INTERPRETER

The GENESIS Interpreter provides a framework in which programmers can specify the control flow of a simulation. The interpreter functions as the main loop of any simulation built on the GENESIS platform. In effect, the *main()* function of a standalone program is replaced by the interpreter. The functions that would normally be called through a compiled *main()* can now be invoked from the interpreter. These instructions can be entered directly from the keyboard at runtime when a programmer is developing a simulation or they can be read from script file. The script file can therefore be used to recreate a specific simulation on demand. By moving this responsibility out of the realm of compiled code and into the realm of a runtime interpreter a great deal of flexibility is gained. Data structures and variables can be created, examined or modified "on the fly" and new functionality can be added to a program without recompilation or leaving the context of a running application. The flow of a typical GENESIS simulation through the interpreter is shown in the diagram below.



## 2.1 INTERPRETER CONTROL CONSTRUCTS

A number of basic control constructs have been built into the GENESIS interpreter to assist programmers in the creation of control scripts for their simulations. These constructs give the interpreter functionality similar to that of UNIX shell scripts. Examples of a few constructs include the ability to define both local and global simple variables (integers, floats, and strings), arithmetic operations, for loops, if-then-else statements, while statements, and foreach statements. A partial list of constructs built into the interpreter along with two simple interpreter "scripts" is shown in the table below.

int float str if-then-else while for loops	<pre>int i = 5 int j str foo = "This is a string variable" echo "foo is set to " {foo} echo "and int i = "{i} for (j = 0; j &lt; 10; j=j+1)   echo "j is currently " {j} end</pre>	<pre>float k = 10.0 while (k &gt; 0)   if ( k / 2.0 == 4.0)     echo "k is currently 8"   else     echo "k isn't currently 8"   end end</pre>
---	--	---

Figure C: A partial list of interpreter constructs and two example scripts

## 2.2 INTERPRETER FUNCTION CONSTRUCTS

The interpreter also has the ability to define functions which contain groups of commands. These functions, once defined, can be called from within the interpreter. Functions take on a similar appearance to functions in a C program. They can take a number of calling arguments and have the ability to return a value to the caller. Scope of simple local variables applies to interpreter functions in the same manner present in C. An example of an interpreter function is illustrated below.

```
//
// GENESIS example to call an interpreter "function". This example just passes in a string
//
function do_it(foo)
str foo
  echo "The calling function passed in the string "{foo}
end

//
// main routine
//
str = "This is the string which do_it gets passed"
do_it(str)
```

Figure D: An example of a GENESIS Interpreter "function"



## 2.3 ACCESS TO C FUNCTIONS

The interpreter can also call C routines which have been compiled into the executing binary. C routines called from the interpreter are always passed an *argc*, *argv* list of any arguments specified by the calling script. These arguments are all passed as string values. The C routine is responsible for conversion of these strings to their proper types. Values or pointers returned by such C routines will be passed back to the interpreter.

A number of predefined C functions are normally registered with the GENESIS interpreter. The ability to add new C routines which can be executed by the interpreter is central to the design of GENESIS. The GENESIS package uses the C library *nlist()* function to allow users to dynamically bind new C routines to the list of commands executable by the interpreter.

The figure below shows a small C function which takes two arguments and returns their sum. This C routine can be compiled and linked to the GENESIS libraries to produce a binary. The routine can then be added to the list of C routines accessible to the interpreter using the interpreter *addfunc* command. Note that the arguments passed from the GENESIS interpreter to the C routine will be in *argc*, *argv* format, with each *argv* passed as a string. It is assumed that the C routine *my\_add()* has been compiled with the GENESIS libraries and is thus present in the running binary.

GENESIS SCRIPT	C-CODE
<pre>// // GENESIS script to addfunc a new C // function, call it, and print the value // it returns // addfunc my_add my_add int  int r r = my_add(2,3) echo "my-add returned "{r}</pre>	<pre>/*  * C function to add two numbers  */ int my_add(argc, argv)     int argc;     char **argv; {     return(atoi(argv[1]) + atoi(argv[2])); }</pre>

Figure E: A GENESIS script which dynamically adds the C routine *my\_add()* and calls it

## 2.4 INTERPRETER CALLING ORDER

When GENESIS interprets a command it first checks to see if the command is a known interpreter construct. If the command isn't a construct it is compared to the list of current interpreter functions. If this comparison fails, a list of known C routines is then checked. If all of these fail, the command is passed on to a UNIX shell. This four-tiered method enables a programmer to take full advantage of the interpreter language, and customized C routines, as well as any binaries on the local UNIX machine.

### 3.0 GENESIS ELEMENTS

In addition to the basic data types available for variable storage in the interpreter GENESIS supports user defined objects called elements. An element contains a user defined data structure and a set of compiled routines for carrying out actions on the contents of the data structure. The contents of these elements are interactively accessible to the user through interpreter-level commands which can display or change data structure contents as well as cause an element to carry out a particular action.

### 3.1 PROTOTYPES VS INSTANCES

Within the interpreter users can create instances of a particular element which is selected from a list of prototype elements. The prototype contains the instructions needed to create an instance of a given type of element. The instance of a particular element is assigned a path name within the tree hierarchy and is allocated a unique data structure.

### 3.2 ELEMENT HIERARCHY

In order to provide a consistent means for accessing elements within GENESIS, elements are maintained in a hierarchical tree structure. All instances of elements created by the GENESIS interpreter require the specification of a path name. All future reference to a substantiated element are accomplished using this path name. Path names follow the conventions used by the UNIX file system. The tree structure imposed on element names does not represent the pattern of functional interconnections between elements, it is simply a tool for organizing groups of elements in the simulation.

A series of GENESIS commands exists to assist a user in traversing the list of instantiated elements. The *le* command is similar to the UNIX *ls* command. It is used to produce a list of the elements attached to a specified element. The *ce* command is analogous to the UNIX *cd* command. It is used to change the current working element to another element in the hierarchy. Finally, the GENESIS *pwe* command performs a similar function to the UNIX *pwd* command, printing out the path name of the current working element. Figure F illustrates a sample hierarchy of elements which might exist in a common GENESIS simulation.

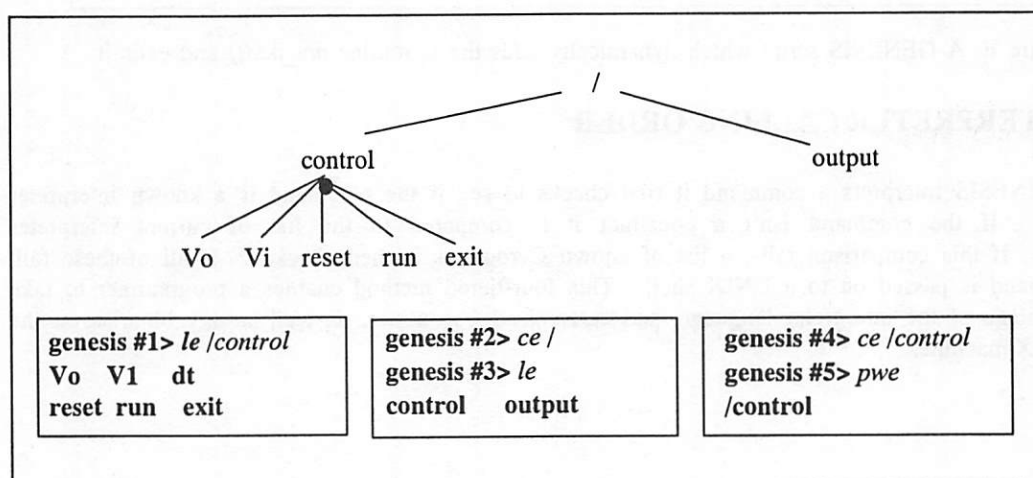


Figure F: An example element hierarchy.  
Three examples of traversing the hierarchy.

### 3.3 GENESIS COMMANDS THAT MANIPULATE ELEMENTS

Three basic GENESIS commands exist for the manipulation of elements. The GENESIS *create* command can be used to acquire a new instance of a known prototype element. All *create* commands require the specification of the prototype or class of the element which is to be created as well as a path name to associate with the new instance of the specified element.

The GENESIS *show* command can be used to print out selected fields of an element. This command is especially useful when trying to debug a program during runtime, allowing the programmer to examine the contents of any GENESIS element while the program is actually executing.

The GENESIS *set* command can be executed from the interpreter to change the current value of any field of an element. This enables simulation users to dynamically change elements while a simulation is executing. This is especially useful when the element in question represents some aspect of a graphical object, such as the width of a form displayed on the screen, because it makes the adjustment of such objects an interactive and speedy process.

Figure G, below, illustrates the use of these commands by creating a new instance of a sample element of class "neutral", showing the current value assigned to the "x" parameter of this element, and setting this field to a new value.

### 3.4 PRE-DEFINED PROTOTYPE ELEMENTS

A wide variety of prototype elements useful for the generation of neural network simulations is distributed with the basic GENESIS system. In addition to computational elements an assortment of XODUS elements, used to represent various X11 widgets within the simulator environment, are also present. Figure H lists the prototype elements currently distributed with the GENESIS package. All of the XODUS elements begin with the letter "x".

```
GENESIS #1 > create neutral /neutral_element
OK

GENESIS #2 > show /neutral_element x
[/neutral_element ]
x      =      0

GENESIS #3 > set /neutral_element x 51
OK

GENESIS #4 > show /neutral_element x
[/neutral_element ]
x      =      51
```

Figure G: Creation, examination, and modification of an element

### 3.5 XODUS ELEMENTS

For each type of widget a corresponding GENESIS prototype element has been defined. These elements contain information about the widgets which might be useful to a simulation programmer. Examples of this information include the current coordinates and dimension of the widgets, the current string stored in a dialog widget, or the current state of a toggle widget.

Ca_concen	efield	pulsegen	xcell
PID	expthresh	random	xconn
RC	freq_monitor	script_out	xdialog
asc_file	funcgen	sigmoid	xdraw
axon	graded	site	xelmtree
axonlink	hh_channel	spike	xfileview
channelA	leakage	symcompartment	xform
channelB	linear	synapse	xgraph
channelC	manuelconduct	synapseA	xlabel
channelC2	nernst	synapseB	xshape
compartment	neutral	unit	xtoggle
defsynapse	periodic	vdep_channel	xview
diffamp	playback	vdep_gate	xviewdata
disk_out	print_out	xbutton	

Figure H: Distributed prototype elements

### 3.6 OBJECT-ORIENTED EFFECT

The interpreter treats XODUS elements as it would any normal element. Requests to create, examine, or modify fields of these elements will be allowed by the interpreter. Since these elements actually represent X11 objects, some of which may be visible on a user's screen, it is critical that the changes made to these elements be reflected in the graphical objects which they represent. The C routines associated with each XODUS prototype element, and thus with each instance of an XODUS element, are responsible for the creation and modification of both the GENESIS element data structure as well as the creation and modification of the X11 widget which this data structure mirrors.

It is equally important that any changes made directly to a graphical object be propagated to the GENESIS element representing that object. For example, changes made by a user to a dialog widget should automatically propagate to the string value stored in the element representing that widget. Along similar lines, the width and height of a widget which has been resized are propagated to the width and height fields of the element representing the widget in the GENESIS hierarchy.

If the propagation of change described in the above paragraphs is implemented, an object oriented extension to X11 is realized. Changes made through the interpreter by a script or by the user propagate to X11 widgets. Changes made to X11 widgets propagate back to the correct GENESIS element which can be interactively examined using the interpreter. This object-oriented effect is illustrated in figure I.

The implementation of this process, given the structure of the GENESIS platform, is quite simple. Propagation of information from GENESIS elements to XODUS widgets is accomplished by defining the appropriate X11 requests in the C code associated with each XODUS prototype element. For example, a request to *create* a new instance of a form widget (element of class "xform") will allocate and attach an instance of the requested prototype to the GENESIS hierarchy and will also execute the necessary C code (*XtSetArg()*'s and *XtCreateManagedWidget()*) to produce a new widget on the current display. Requests to *set* fields of an XODUS element will result in both the updating of the instances field as well as an appropriate series of C calls to change the corresponding widget (*XtSetValues()*).

To assure propagation of widget changes to elements, the GENESIS package takes advantage of the X11 Callback mechanism. Callbacks are registered with each widget created via the GENESIS package, guaranteeing that the proper element is notified of any important changes to an XODUS widget.

### 3.7 CREATING NEW "PROTOTYPE" ELEMENTS

The power of the GENESIS package would be severely limited if provisions had not been made to allow simulation programmers to extend the set of prototype elements. It is critical that the addition of new prototype elements into the GENESIS package be cleanly and easily integrated with the interpreter and data structure handlers described above.

All GENESIS elements begin with a common structure which enables the interpreter and element handler routines within the GENESIS package to manage instances of the element. Following this common structure, each prototype element is free to define any additional fields needed to define the new element.

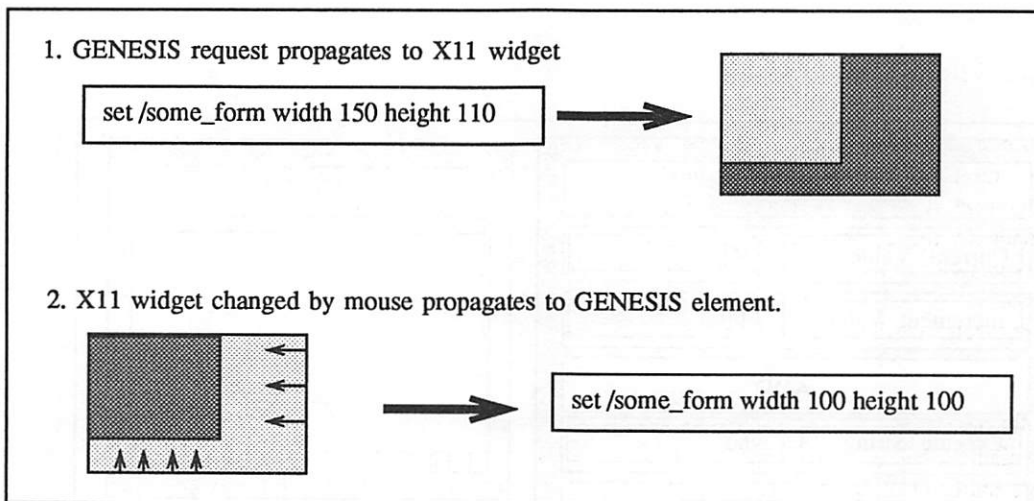


Figure I: In (1) an interpreter request to change the width and height fields of an element propagates to update the displayed widget. In (2) the user has used a mouse to resize a widget and the new width and height is propagated back to the element which represents the widget in the GENESIS hierarchy.



In addition to the new "data structure" which defines the prototype data fields, a series of C routines must be written to handle input, processing, and output for the new element. This includes, but is not limited to, routines which handle the creation of new instances of the prototype, and routines which handle requests to set fields in an instance of the element and propagates those changes to any other objects associated with the field (eg: X11 objects).

#### 4.0 AN EXAMPLE SIMULATION

A simple example of an "application" developed using the GENESIS platform is shown in figure J. Users can enter values for "Current value" and "Increment value". "Current value" is incremented each time the "ADD" button is clicked on. This action will also graph the new "Current value" in the displayed graph. The "SHELL FUNCTION" button will pass the string specified in the "Execute String" dialog to a UNIX shell. Finally, the "C ROUTINE" button will pass the current value of the "Execute String" dialog to the C routine *print\_backwards()*. This is just a simple C routine which prints the string defined by *argv[1]* in reverse.

The GENESIS interpreter control script used to implement the graphical interface and functional application shown in figure J is illustrated in figure K.

#### 5.0 AVAILABILITY

A full source distribution of the GENESIS package is currently available from the California Institute of Technology. Request for distribution forms can be obtained via anonymous ftp from [genesis.cns.caltech.edu](ftp://genesis.cns.caltech.edu). A distribution fee of \$250.00 to cover cost of media and hardcopy documentation is currently in effect. The distribution includes full source, several example simulations, a manual on the GENESIS simulator, and the creation of an ftp account on [genesis.cns.caltech.edu](ftp://genesis.cns.caltech.edu) from which future releases and additional packages built with the GENESIS platform can be obtained at no charge.

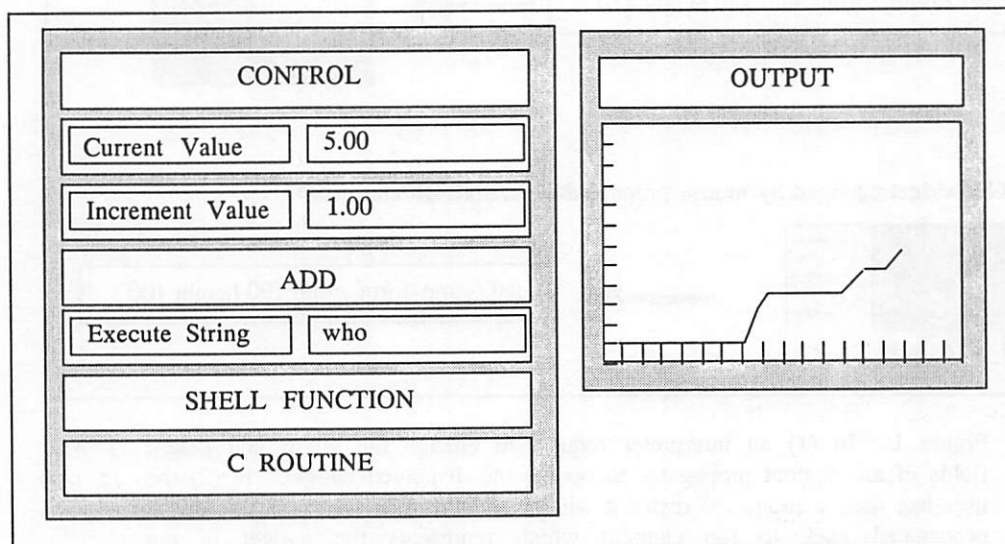


Figure J: A sample "application" implemented in GENESIS

```
//
// GENESIS Script to implement sample application
//
addfunc print_backwards print_backwards
int x = 0
create xform /control
create xdialog /control/current -title "Current value" -value 5.0
create xdialog /control/increment -title "Increment value" -value 1.0
create xbutton /control/add -title "ADD" -script do_add
create xdialog /control/string -title "Execute string" -value who
create xbutton /control/exe -title "SHELL FUNCTION" -script do_exec
create xbutton /control/print -title "C ROUTINE" -script do_print
create xform /output
create xform /output/graph
xshow /control
xshow /output

function do_add
  set /control/current value { get(/control/current,value)+get(/control/increment,value)}
  addpts /output/graph x {get(/control/current,value)}
  x = x + 1
end

function do_exec
  sh {get(/control/string,value)}
end

function do_print
  print_backwards({get(/control/string,value)})
end
```

Figure K: The interpreter script to create and execute the application in figure J.

## 6.0 CONCLUSION

The development of the GENESIS platform illustrates the power which can be gained by replacing the conventional *main()* function of an executing binary with an interpretive front end. The ability to dynamically and interactively create, modify, and examine elements used by an application, as well as the object-oriented properties which can be generated when such a system is properly implemented, greatly enhance the speed and ease with which applications can be developed and debugged. This, coupled with the ability to quickly and interactively define and customize graphical interfaces, illustrates one possible implementation of a powerful development tool which should be useful in a wide range of applications.

## 7.0 ACKNOWLEDGEMENTS

This research was supported by the NSF (EET-8700064), the NIH (BNS 22205), the ONR (Contract N00014-88-K-0513), the Lockheed Corporation, the Caltech President's Fund, the JPL Director's Development Fund, and the Joseph Drown Foundation.



**Matt Wilson**  
*Cal Tech*

Matt Wilson is a graduate student in the Computation and Neural Systems program at the California Institute of Technology. In the course of his research in neurobiology Matt served as the primary author of the GENESIS platform. Matt received his undergraduate degree in Electrical Engineering from Rensselaer Polytechnic Institute in Troy, NY and his Masters degree in Electrical Engineering from the University of Wisconsin, Madison.



**Upinder Bhalla**  
*Cal Tech*

Upinder Bhalla is a graduate student in Biology at the California Institute of Technology. In course of his research in neurobiology he played a central roll in the development of the XODUS graphical interface. Upinder received his undergraduate degree in Physics at Cambridge University, England.



**John Uhley**  
*Cal Tech*

John Uhley currently works for the Division of the Computation and Neural Systems at the California Institute of Technology on the software development team for the GENESIS package. He played a central role in the development of the XODUS interface for the GENESIS package. John received his degree in Computer Science from the University of California at Berkeley.



**David Bilitch**  
*Cal Tech*

David Bilitch graduated from the University of Southern California with a bachelors degree in Computer Science. He is currently on staff at the California Institute of Technology. Dave was responsible for the implementation of the GENESIS Script Language Interpreter.



**Mark Nelson**  
*Cal Tech*

Mark E. Nelson is a senior research fellow in the Computation and Neural Systems program at the California Institute of Technology. Mark received his Ph.D. in Physics from the University of California, Berkeley.

**James Bower**  
*Cal Tech*

James M. Bower is an assistant professor associated with the Computation and Neural Systems program at the California Institute of Technology. He received his graduate degree at the University of Wisconsin, Madison. He was a postdoctoral trainee at the New York University Medical Center.





# Keynote - A Language and Extensible Graphic Editor for Music

*Tim Thompson*  
*AT&T Bell Laboratories*  
*Holmdel, New Jersey*  
*tjt@twitich.att.com*

## Abstract

Keynote is a programming language for manipulating and generating music with MIDI-compatible equipment. It was designed for and in the style of the UNIX<sup>®</sup> software system - an application-specific "little language" and interactive shell. Most obviously used for algorithmic music composition, Keynote also serves as a more general utility for realtime and non-realtime MIDI data manipulation. By adding only a few built-in functions to the language, a graphic interface was recently added to Keynote. This built-in graphic interface did not, however, build-in any particular user interface. All of the nested pop-up menus and operations of a graphical music editor have been implemented in the Keynote language itself. The result is an extensible tool, similar in spirit to the Lisp-based extensibility of emacs, and easily modified and enhanced by end users.

## Introduction

Most professional and amateur musicians use MIDI (Musical Instrument Digital Interface) equipment with personal computers such as the Macintosh<sup>®</sup> and MS-DOS<sup>®</sup>-compatible PCs. MIDI interfaces are rarely seen on UNIX systems, but it seems inevitable that these two disjoint worlds, separately based on wildly successful standardizations, will soon come together. Already, UNIX systems with MIDI can be easily and cheaply built with off-the-shelf hardware (ie. 386-based computers and MPU-compatible interfaces). The music software for these systems should not be limited to a porting of the existing PC software - doing so would ignore the UNIX software tool philosophy and its considerable benefits. Some UNIX software tools for MIDI have been developed, as specific algorithmic programs<sup>[1]</sup> and small tools<sup>[2]</sup>. However, there have been no reports of a power tool that has the straightforward flexibility of awk and the extensibility of emacs - Keynote fills that void.

## Background and History

A large variety of music software is available for personal computers, much of it well-written and well-featured. Those features, however, are usually fixed, and active users of a software package will inevitably find a need for features that are not already provided. The editing operations of sequencers (software that allows the entry, editing, and playback of music) are notorious for this kind of limitation - there are typically dozens of editing commands, yet a user will easily encounter situations in which a desired operation is difficult (requiring many steps to accomplish) or impossible. Music software marketed as "algorithmic composition"

---

<sup>®</sup> UNIX is a registered trademark of AT&T. Macintosh is a registered trademark of Apple Computer, Inc.  
MS-DOS is a registered trademark of Microsoft Corporation.

is, somewhat ironically, also limited - the algorithms are those of the original developer, not the user, and although there may be many algorithmic parameters that can be changed, the fundamental algorithms are fixed (and sometimes secret). Of course, all these built-in limitations guarantee a perpetual market for software upgrades - the next version always has more features and will be available Real Soon Now.

Keynote is a tool designed to avoid such limitations. The UNIX system deserves more flexible music software, and one of the best routes to flexibility is through programmability. Application-specific "little languages" are a time-honored tradition for UNIX software tools - `awk`[3], `pic`[4], `S`[5], `pico`[6], and `squeak`[7] name only a few. So, Keynote was designed as a musical "little language" specifically tailored for use with MIDI equipment. Development began over 3 years ago, and the first version was little more than a crude BASIC-like language. Two years of development produced a language that was mature and expressive - it had undergone three iterations in design and implementation. However, it was a language only, and lacked the graphical interface that makes most commercial products appealing and convenient to use. The last year has been spent designing and implementing graphical additions to the language, and then using those additions to build a user-extensible music editor. This development path has been extremely effective; the new graphical interface has been able to leverage its underlying programmability in numerous and surprising ways, as examples will show.

## Related Work

The concept of a programming language specialized for music is not in itself particularly innovative or unusual. There are many examples in computer music research<sup>[8][9][10]</sup>, although many of these languages are intended for generation of audio waveforms directly and are not useful for MIDI work. Also, most of these languages are far from conventional, and although this is probably a purposeful trait of adventurous research efforts, it limits their potential applicability for day-to-day use by normal users. For example, imagine if `awk` were patterned after APL instead of C. Would it be anywhere near as popular? The UNIX system itself would not be as popular if it were not as conventional as it is elegant and flexible. So, perhaps the most unusual aspect of Keynote as a language is its conventionality. That does not belittle its other unique features, but merely emphasizes that they are accessible within a framework that is immediately familiar.

Several languages intended for use with MIDI on personal computers are being sold commercially<sup>[11]</sup>, and again they are often unconventional. Some of them<sup>[12][13]</sup> have interfaces in which the programming language itself is graphical - the user creates a flow chart of modules and data paths between them. Although interesting and useful for smaller applications, these non-textual programming languages quickly become a limitation when doing larger projects. The language most similar to Keynote is Ravel<sup>[14]</sup>, a marginally C-like language whose most distinctive feature is the ability to have concurrently-executing functions that interact with each other.

The graphical editor built out of Keynote is comparable in editing capability to a typical PC music sequencer. The difference, however, is that it is completely user-defined. Every operation can be customized to suite the taste of a user, and new editing operations are easily added. Of the many commercial sequencer/editor packages, only 1 is currently programmable - Personal Composer<sup>[15]</sup>. Though it has a reputation for being buggy, Personal Composer is also admired for its concept and potential; it is programmable via macros and a built-in Lisp interpreter. Since Personal Composer has been available for many years, it is somewhat surprising that no other extensible products have appeared in that time. However,

people in the industry still realize the power that user-extensibility holds<sup>[16]</sup>, and are looking forward to such products. Indeed, music languages are being introduced in the newest versions of the Cakewalk<sup>[17]</sup> and Dr. T's KCS<sup>[18]</sup> software. Such "add-on" languages are not likely to work as well or be as useful as a language designed into a product from the beginning. For example, they have lost the advantage of having the language available as a tool during their own development. Keynote was a mature language before the graphic interface was even considered, and the subsequent implementation of the graphic interface was *considerably* easier because so much of its functionality could be done with Keynote code.

## The Keynote Language

The Keynote language is designed for convenient and straightforward expression of musical algorithms. It will be immediately familiar to users of the UNIX system - it is very similar in style to awk. Variables need not be declared, and types are determined by their use. The data types are integer and floating-point numbers, strings, and musical phrases. Arrays are associative (ie. index values can be strings), and can be passed by reference to functions. There is a full set of operators and control structures (except for `switch`). `#include` and `#define` work as in the C pre-processor. User-defined functions can have arguments and return values of any type. Local variables can be provided in a function by including them in the definition as extra parameters. When first referenced, functions are automatically loaded from library files, with path searching. Functions can be redefined on-the-fly by reloading them, a great convenience for interactive development.

Naturally, the language has features designed specifically for manipulation of MIDI data. Musical phrase variables contain MIDI data - complete notes, isolated note-on's, isolated note-off's, and arbitrary MIDI bytes. Operators work on these phrases in a natural fashion for merging, modification, and selection. The *select* operation is particularly expressive; it picks out all notes for which an arbitrary condition is true. Control structures can loop through phrases, and conditionals can test for inclusion of notes in phrases. The syntax of C structure elements is used to refer to the attributes (pitch, volume, channel, etc.) of a musical phrase. The index of an associative array can even be a musical phrase. Some random examples to illustrate the style and syntax:

```
p.input = "jsbach.k"      # Read a phrase from the library.
p2 = p { ?.dur < 1b }     # Select notes with duration < 1 beat.
p = p - p2                # Remove those notes from p.
p2.time += 1b             # Delay all notes in p2 by one beat.
p2.chan ++                # Increment channel of all notes in p2.
p = p | p2                # Add notes of p2 to p (in parallel).
p += p                    # Repeat the phrase, in series.
result = ''               # Initialize an empty phrase.
for ( nt in p ) {         # Loop through all notes in p.
    nt.time = p.length     # Reverse the starting time of the note
    - nt.time - nt.dur    # within phrase p.
    result |= nt           # Add it to the result.
}
result.output = "result.k" # Write it to a file.
```

Since a phrase variable can contain arbitrary MIDI data, Keynote has no problem manipulating system exclusive and other non-note data. There is also a convention for embedding "text" notes within a phrase. These special notes can be used as a hook for embedding Keynote statements *within* musical phrases, a feature that has immense potential.



For example, here it is used in a phrase constant to embed a tempo change:

```
'a,b,c,"Tempo=400000",d,e,f'
```

The user-defined function that normally plays phrases will automatically scan for such text notes and schedule appropriate actions to control the tempo. Text notes have also been used for embedding phrase expressions within phrases:

```
'a,b,c,"{reverse(ph)}",d,e,f'
```

Again, the function that plays phrases can automatically scan for such text notes, evaluate the expressions they contain, and incorporate the results into the final phrase before actually playing it. This is essentially a way of delaying the evaluation of statements, and forcing them to be evaluated on demand. Nesting is possible, for example phrase *ph* in the example above could itself contain text notes with phrase expressions. Text notes are a powerful feature, and will likely find other use.

### Algorithmic Examples

Many useful operations can be expressed with small amounts of Keynote code. Transposing a phrase by an octave is a single statement, and can be packaged as a one-line filter:

```
key -c 'p.input="|" ; p.pitch+=12 ; print p'
```

The `-c` option of `key` (the Keynote interpreter) allows programs to be put on the command line. This program has 3 statements - the first reads a phrase from standard input, the second adjusts the pitch of its notes, and the third sends the result to standard output. Here is a "limiter" that transposes down all notes whose pitch is greater than some limit:

```
key -c 'p.input="|"; a=p{?.pitch>100}; p=p-a; a.pitch-=12; print p|a'
```

The second statement in this example is a *select* operation that picks out all notes whose pitch is greater than 100 (MIDI pitch values range from 0 to 127). The third statement removes those notes from the original phrase, the fourth statement transposes them, and the last statement merges them back into the final result. As you can see, the specialization of the language allows a concise expression of typical operations. This then makes it easier to build larger, more complex operations. It also makes it easier to experiment, prototype, and get results quickly - an important advantage for algorithmic composition, where hearing the results of an algorithm is an intimate part of the design process and is dominated by the time to express the algorithm.

*Markov chains*<sup>[19]</sup> are often used in algorithmic composition. One application of this technique uses an existing piece of music to initialize a transition table, which is then used to generate a new piece of music that sounds "similar" to the original. The similarity is dependent on the *order* of the chain - each event in an *N*-th order Markov chain depends on the *N*-1 previous events. A Keynote program to generate *N*-th order Markov chains can be written in only a few dozen lines of code, shown in the appendix of this paper. This example is greatly simplified by associative arrays in which the indicies are musical phrases.

### Realtime Capabilities

When Keynote is used as an interactive shell, phrase expressions are immediately evaluated and played in realtime via MIDI output, allowing convenient experimentation and immediate feedback. Keynote also has more general realtime capabilities. Whenever MIDI output is being generated, Keynote enters a mode during which:

- Phrases can be scheduled to be played at specific times, once or repeatedly.
- User-defined functions can be scheduled at specific times, once or repeatedly.
- Interrupts can trigger the invocation of user-defined functions. Interrupts can be generated by the pressing of a console key, mouse activity, or the arrival of a MIDI input message (e.g. the note-on message when a key is depressed).
- Scheduled phrases and functions can have an associated *tag*, allowing them to be de-scheduled.
- MIDI input can be recorded and assigned to a phrase variable.
- MIDI input can be merged into the MIDI output.

These features can be used to build a variety of interesting interactive toys - an echo effect, "auto-chording", even the trading of improvised riffs between human and computer. Since Keynote is an interpreted language, there is a limit to the amount of real-time processing that can be done before delays become noticeable. Still, for many realtime applications it is perfectly acceptable, and for the others it is a convenient prototyping tool.

### Realtime Examples

An "echo" program - for each note received on MIDI input, echo it some time later to MIDI output - was the first test for the realtime capabilities of Keynote. It was important that something that was simple to explain be simple to express. Among other things, this introduced the requirement that Keynote be able to treat note-on's and note-off's independently. Here is the complete source for an echo program:

```
function echoit(a) {
    sched(a,1b)      # hard-coded echo time of 1 beat
}
function echo() {
    interrupt(echoit,NOTEON | NOTEOFF)
    realtime()
}
```

Calling the `echo` function begins the effect, which continues until a console key is pressed (the default way in which the `realtime` mode is terminated). The `echoit` function is called whenever a MIDI note-on or note-off is received, and schedules the playing of the echo. A more elaborate version allows the echo time to be varied and interactively requests the user to specify (via MIDI input) the region of notes to be echoed:

```
function getanote() {
    interrupt(gotanote,NOTEOFF)
    realtime()
    return (Got)
}
function gotanote(a) {
    Got = a.pitch
    stop()
}
function echoit(a) {
    if ( a >= Echolow && a <= Echohigh )
        sched(a,Echotime);
}
```



```

function echo(tm) {
    if ( nargs() != 1 ) {
        print "usage: echo(echo-time)"
        return
    }
    print "Press the 2 notes of the echo range..."
    Echolow = getanote()
    Echohigh = getanote()
    Echotime = tm
    interrupt(echoit,NOTEON | NOTEOFF)
    realtime()
}

```

The `getanote` function in this example shows another (essentially non-realtime) use of the realtime mode - it waits for the user to press a note on the MIDI keyboard and returns the value. The `gotanote` function is called when a MIDI note is received, the `stop` function terminates the realtime mode, and `getanote` returns the note's value.

## The Graphical Interface

Keynote originally had only a textual interface - a programming language and interactive shell. Musical algorithms were easily expressed, but it lacked a convenient graphical interface for interactive editing. Such an interface could have been added as a separate process, with Keynote serving in parallel as a programmable utility. However, a programmable graphical interface is as useful and interesting as a programmable music processor, and there is considerable synergy when combining them. So, graphical extensions were added to the Keynote language itself.

The first step was choosing a style for displaying the music. The two common alternatives are standard music notation (the style used in conventional sheet music) and "piano-roll" style (the horizontal axis is time, the vertical axis is pitch, and notes are displayed as boxes whose length shows the duration of the note). Standard music notation is extremely difficult to do well, and introduces too many representational problems unrelated to MIDI data, so Keynote uses the easier and more straightforward piano-roll style. Standard MIDI Files (an industry-wide standard) can be used to transfer music from Keynote to other software packages that can generate standard music notation.

The new Keynote graphical interface looks every bit like a "graphical editor", however that impression is due as much to the default user-level customization as it is to the newly added features in the language itself. The purpose of the language extensions was to add *only*: the ability to display musical phrases in piano-roll style, the ability to manage nested pop-up menus whose items can invoke user-defined functions, and the ability to detect mouse activity and invoke user-defined functions. The extensions were implemented with only 15 built-in functions and several dozen special global variables. The functions fall into 3 categories:

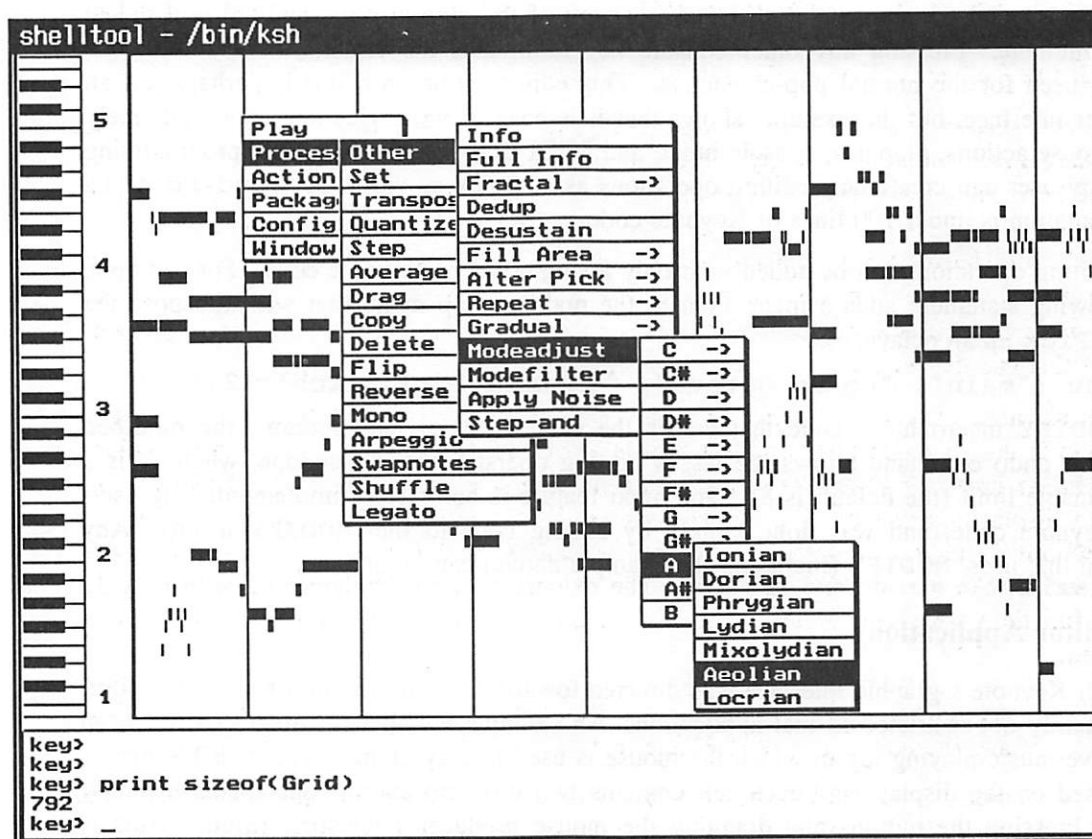
- |                   |   |
|-------------------|---|
| <b>Display</b>    | <code>graphics</code> , <code>draw</code> , <code>redraw</code> - These functions control the graphics display and the drawing of phrases and lines.  |
| <b>Mouse</b>      | <code>menu</code> , <code>action</code> , <code>setmouse</code> , <code>sweep</code> - These functions establish the contents of pop-up menus, and allow mouse actions to trigger the execution of user-defined functions.                    |
| <b>Efficiency</b> | <code>drag</code> , <code>gridsweep</code> , <code>mouseinpick</code> , <code>setpick</code> , <code>gridadd</code> , <code>gridsub</code> , <code>gridpan</code> , <code>flash</code> - These functions were actually implemented with user- |

level Keynote code when first prototyped, but have been implemented as built-in functions to improve interactive performance.

These extensions were easily used to build a graphical editor, but they are not restricted to it. So, the term "graphical interface" is often purposely used to emphasize the more general nature of the extensions.

## The Graphical Editor

The entire user interface of a graphical music editor has been implemented with Keynote code. This includes all pop-up menus and the several dozen editing operations they invoke. The user interface is centered around the display and editing of a musical phrase - the `Grid` variable. The screen is split into two windows: a textual window in which the normal interactive command interpreter is run, and a graphical window in which the `Grid` phrase is displayed and edited. The left mouse button is used to sweep out groups of notes, selecting them for editing; the currently-selected notes are called the `Pick` phrase. The right mouse button is used to access nested pop-up menus of editing operations that transform the notes of the current `Pick`. The transformed notes become the new `Pick`, so that a sequence of operations can be done quickly without having to reselect the notes. The pop-up menus also control windowing, configuration settings, and other activities not related to the current `Pick`. The following is an example of the screen display, where a deeply nested menu item is about to be selected.



Some of the more interesting operations in the editor show the flexibility of Keynote's user-programmable graphical interface. A good example is the "flashing" of notes. When a

group of notes is selected and the `Play` menu item is invoked, the notes will flash off and on as they are played via MIDI output. This could easily have been built into the language, just as real-time MIDI output is built into the language. However, recall that Keynote can schedule the invocation of user-defined functions. So, to prototype the flashing notes it was easy to schedule the erasing and redrawing of individual notes at the appropriate times. Surprisingly, the performance of this prototype (using less than 50 lines of Keynote code) was perfectly acceptable, eliminating the need to change the language. More surprisingly, its behavior is even better than a built-in solution. Since user-scheduled actions are interleaved with and give priority to MIDI output, the flashing of notes in a busy musical passage can lag behind the MIDI output - the result is accurate output timing in spite of the flashing notes. A user-level solution is also more flexible. One user didn't like having the notes erased for the entire duration of each note - he wanted just a quick flash. A one-line change gave him the desired behavior.

Another interesting example is the `Step-and-Edit` menu item that lets a user step through the notes of the current `Pick`, interactively changing their pitch and time. It is a convenient mechanism because everything is controlled from the mouse; the right button plays (via MIDI output) the next note, and the left button plays the previous note. While a note is playing, ie. while a mouse button is depressed, the mouse can be moved up and down, dragging with it the pitch of the note. Each time the pitch changes, it is reflected both on the graphic display and in the MIDI output. So, you can go back and forth from note to note trying different pitch intervals, using only the mouse. Several console keys can be pressed for special operations: 'c' will make a copy of the current note, and 'd' will delete the current note. Pressing any other console key terminates the effect, so the mouse can again be used for the normal pop-up menus. This editor-within-an-editor is perhaps not an ideal user interface, but the example shows that it is easy to write Keynote code that freely mixes mouse actions, graphics, console input, and MIDI I/O. With only a little programming effort, any user can create new editing operations as complex as the `Step-and-Edit`; its implementation is under 100 lines of Keynote code.

Some editing operations can be added with only a single line of Keynote code. For example, the following statement adds a menu item to the main pop-up menu that will transpose the current `Pick` up an octave:

```
menu ("main", "Up an Octave", "{MODIFY(Pick.pitch+=12)}" )
```

The `MODIFY` macro hides code that erases the current `Pick` and redraws the modified notes. An undo command allows the last  $N$  editing operations to be undone, where  $N$  is a user-definable limit (the default is 8). The undo feature is completely implemented by user-level Keynote code, and was done largely by adding code to the `MODIFY` macro. Any operation that uses `MODIFY` (including the example above) can be undone.

## Non-Editor Applications

Although Keynote's graphic interface was directed toward the construction of a music editor, it is certainly not restricted to that application. An example is called `Mouse Matrix`, an interactive music-playing toy in which the mouse is used to play chords. An invisible matrix is imposed on the display, and each cell contains two different chords, one for each mouse button. Pressing the buttons and dragging the mouse produces interesting results. This is only a small example of the non-editor applications that might be constructed.

## Portability

Keynote is highly portable. Proper realtime operation requires:

- The ability to quickly get and put single MIDI bytes.
- The ability to quickly poll for pending MIDI and console input.
- A clock accurate to 5 milliseconds or less.

The graphic features of Keynote require:

- The ability to poll the current position and button status of the mouse.
- The ability to draw a line on the screen.
- The ability to copy a raster from the screen into memory, and back. The raster data can be machine-dependent; Keynote makes no assumptions about its format.

Everything is derived from these basic functions. Raster operations are used wherever possible to improve interactive performance. For example, the rasters for pop-up menus are saved, so they can be redisplayed quickly - an extremely important feature for convenient use of pop-up menus. Text is also displayed with raster operations; the font is defined by a human-readable (and hence editable) ASCII file, and the characters are initially constructed on-screen in order to create the machine-dependent rasters. Keynote has been ported to: the AT&T 6386 (using the X Window System with a small device driver that supports MPU-compatible MIDI interfaces), Sun workstations (using SunView™ with an RS232-to-MIDI interface), the AT&T UNIX PC, MS-DOS-compatible PCs, the Macintosh, and the Amiga®. Given good graphics support, porting is fairly easy - the graphic part of the Amiga version was ported in one weekend.

## Summary

A UNIX system used for MIDI music generation and manipulation needs a tool with the flexibility of awk and the extensibility of emacs. Keynote is an example of how it can be done, and illustrates the value of embedded programmability. By adding only a few built-in functions to the Keynote language, it has been possible to build the entire user interface of a graphical music editor with Keynote code. The result can easily be enhanced and extended.

---

® Amiga is a registered trademark of Commodore-Amiga, Inc.  
SunView is a trademark of Sun Microsystems, Inc.



## REFERENCES

1. Peter S. Langston, (201)644-2332 or *Eddie & Eddie on the Wire: An Experiment in Music Generation*, USENIX Conference Proceedings, Summer 1986.
2. Michael Hawley, *MIDI Music Software for UNIX*, USENIX Conference Proceedings, Summer 1986.
3. A. V. Aho, B. W. Kernighan, P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, 1988.
4. B. W. Kernighan, *PIC - A Language for Typesetting Graphics*, Software Practice and Experience, Vol. 12, No. 1, 1982.
5. R. A. Becker and J. M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth Advanced Book Program, 1984.
6. Gerard J. Holzmann, *Beyond photography: the digital darkroom*, Prentice-Hall, 1988.
7. L. Cardelli and R. Pike, *Squeak: A Language for Communicating with Mice*, Computer Graphics, Vol. 19, No. 3, 1985.
10. Bill Schottstaedt, *Pla: A Composer's Idea of a Language*, Computer Music Journal, Vol. 7, No. 1, Sprint 1983.
10. C. Fry, *Flavors Band: A Language for Specifying Musical Style*, Computer Music Journal, Vol. 8, No. 4, Winter 1984.
10. Xavier Rodet and Pierre Cointe, *FORMES: Composition and Scheduling of Processes*, The Music Machine, MIT Press, 1989.
11. C. Scholz, *HMSL Software Language*, Music Technology, September, 1988.
13. Randall Stokes, *The Anything Box*, Music Mind Magic, 401 S. Silver, Centralia, WA 98531.
13. John Dunn, *Music Box*, PO Box 5348, Santa Rosa, CA 95402.
14. James R. Binkley, *Ravel 2.0 - A Music Programming Environment*, 5814 SW Taylor, Portland, Oregon, 97221.
15. Jim Miller, *Personal Composer*, 2449 76th Avenue SE, Mercer Island, WA 98040.
16. Carter Scholz, *Keyboard Magazine*, Guest Editorial, June, 1989.
17. Twelve Tone Systems, Inc., PO Box 226, Watertown, MA, 02272.
18. Dr. T's Music Software, 220 Boylston Street, Suite 206, Chestnut Hill, MA, 02167.
19. Kevin Jones, *Compositional Applications of Stochastic Processes*, Computer Music Journal, Vol 5, No. 2, Summer 1981.



## Appendix: Generating Markov Chains

```

# usage: markov(ph,order,cnt)
#   ph   - the phrase used to initialize the transition table
#   order - the order of the Markov chain
#   cnt   - number of chain links to put in the generated phrase
function markov(ph,order,cnt) {
    markovprep(ph,order)    # Prepare transition table
    return (markovmake(cnt)) # Use it
}

function markovprep(ph,order, chunk,n,nextnt) {
    arrayinit(After)        # Stores chunks-versus-possible-nextnotes
    ph = strip(ph)          # To simplify, all notes are same duration
    chunk = ''              # Starting chunk is the first order-1 notes
    for ( n=1; n<order; n++ )
        chunk |= ph%n       # ph%n is the n'th note of phrase ph
    for ( ; n <= sizeof(ph) ; n++ ) {
        nextnt = strip(ph%n) # strip() removes surrounding space
        # Add the next note to the list of notes
        # that can follow the current chunk
        if ( ! chunk in After )
            After[chunk] = ''
        After[chunk] |= nextnt
        # Advance the chunk, by removing the first note and
        # adding the new note to the end.
        chunk%1 = ''        # remove 1st note
        chunk = strip(chunk) # and space it leaves
        chunk += nextnt
    }
}

function markovmake(cnt, nt,chunk,n,result) {
    # Pick a random starting chunk
    n = rand(sizeof(After))
    for ( chunk in After )
        if ( n-- <= 0 )
            break
    chunk = phrase(chunk)    # undo coercion of indicies to strings
    result = chunk
    while ( cnt-- > 0 ) {
        if ( ! chunk in After ) {
            print "Warning - terminal chunk =",chunk
            break
        }
        # Randomly pick a note that can follow the current chunk
        choice = After[chunk] % (1+rand(sizeof(After[chunk])))
        result += choice
        # Update chunk by removing first note and adding new note.
        chunk%1 = ''
        chunk = strip(chunk)
        chunk += choice
    }
    return (result)
}

```



**Tim Thompson**  
AT&T

Tim Thompson is a Distinguished Member of Technical Staff at AT&T Bell Laboratories in Holmdel, New Jersey. He received his B.S. (1976) and M.S. (1978) in Computer Science from the University of Illinois in Champaign-Urbana. Working for Bell Laboratories and using the system since 1978, he has been involved in computer-aided design for VLSI and high-frequency circuit packs, tools for the design of optical systems, and fax image processing. His free time is often spent in musical pursuits, focused currently on software tools for the system.

# Integrated Interactive Access to Heterogeneous Distributed Services<sup>1</sup>

*Joel S. Emer*

Distributed Systems Architecture and Performance  
Digital Equipment Corporation  
Littleton, Massachusetts  
Internet: emer@dsmail.dec.com

*William E. Weihl*

Laboratory for Computer Science  
Massachusetts Institute of Technology  
Cambridge, Massachusetts  
Internet: weihl@photon.lcs.mit.edu

## Abstract

In this paper, we consider how to provide interactive users with uniform and integrated access to heterogeneous distributed services. Access to distributed services through a single interface has many advantages. One is that it provides the user with an integrated interface environment that allows easy interaction between uses of distinct services. Another is that it simplifies creation of interfaces to new services by obviating the need to create a custom interface for that particular service. As a particular instance of such an interface, we examine the use of editors as an interface to distributed services. To date, the means provided within editors for accessing services has been relatively ad-hoc. In this paper, we examine the more general technique of using remote procedure call as the service access paradigm. This technique is shown to have many advantages including: ease of integration of new services; allowing for more powerful services since they can be written in languages and on hardware platforms appropriate to the service; and allowing access to remote data and access from multiple sites.

## 1 Introduction

In this paper, we consider how to provide interactive users with uniform and integrated access to heterogeneous distributed services. Accessing a wide variety of services from a single user interface has many advantages. One is integration: rather than having a separate interface to each service, users can easily take data returned from one service and send it to another. Another is uniformity: there is uniform availability of sophisticated interface mechanisms such as command and filename completion, as well as a single mechanism for manipulating text (e.g., cutting and pasting or searching). In addition, with a single interface environment, the user can be given the opportunity to customize the environment

---

<sup>1</sup>This work was supported in part by Digital Equipment Corporation, in part by the National Science Foundation under Grant CCR-8716884, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contracts N00014-83-K-0125 and N00014-89-J-1988. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Digital Equipment Corporation, the Defense Advanced Research Projects Agency, the National Science Foundation, or the U.S. Government.

with additional commands or particular settings for built-in commands; by integrating services into the user's existing text manipulation environment, the same customization applies to all services.

As a simple example, suppose a person uses both an online calendar database and an electronic mail facility. Providing an integrated interface to the two services makes it easier for the user to move information from one to the other. For example, an integrated interface makes it easy to cut out a calendar entry for inclusion in a mail message. Similarly, with an integrated interface one might specify the date, time, room, and agenda for a meeting by successively selecting the appropriate parts of a mail message.

Providing a single remote access mechanism makes it easy to create new services and to access them. As we discuss in more detail below, we believe that remote procedure call (RPC) [BN84] provides a significantly better mechanism for accessing distributed services, including *local* services, than other methods. However, not all RPC systems are equally useful in this regard; we discuss below the characteristics of RPC systems that we believe are essential in this application.

As a specific example of an integrated interface to distributed services, we present our experience in adding a powerful RPC capability to the extension language of an editor (in particular, GNU Emacs [Sta86]). We chose Emacs in large part because of the powerful customization capabilities it already provides, as well as because it is widely used and available. We believe that a similar capability could be added to other user interface management systems, with similar effect.

Reference models that include integrated user access to distributed services have been developed [Lan86]. In addition, it is already common to include interfaces to local services in editors, both in Emacs and in other editors (e.g., see [Sta81, Dig85, Wat86]). For example, aids to text entry, such as spelling checkers, are commonly accessed from editors, as are interfaces to text-oriented services such as electronic mail and bulletin boards. Other services that process information generated in editors (and also produce information to be used in editors), such as compilers, are also widely used directly from editors. Our goal in adding RPC to Emacs Lisp (also called Elisp) was to explore whether similar advantages could be obtained with remote services. For example, a bibliographic search facility, or any other database, might be provided on a single site on a network, and accessed via RPC from many other sites. We have found that this additional capability enhances the power of the system significantly, by making it easy to provide users with integrated access to a wide range of heterogeneous services.

In the next section of this paper, we summarize why we believe that RPC is superior to other mechanisms for accessing services from editors or other environments. In Section 3, we discuss the particular capabilities provided by our RPC system, which is called Mercury [LBG\*88], that are important in this application. Next, in Section 4, we describe how we added Mercury RPC to Elisp. Then, in Sections 5 and 6, we describe the results of some preliminary performance measurements, and summarize some of our experience to date in using RPC to add service interfaces to Emacs. Finally, we conclude in Section 7 with a summary of our results.

## 2 Alternative Access Methods

There are two basic techniques for augmenting an editor to provide access to a new service: the editor source code can be modified directly, or a program can be written in an *editor extension language*. Modifying the editor source has obvious disadvantages: it generally requires relinking the editor to incorporate the changes, and it leads to a horrendous version management problem, as different users incorporate different services. Thus, many editors today provide at least primitive capabilities for extension that do not involve modifying the editor source.

Regardless of whether the editor source is modified or a program is written in some extension language, we have a choice to provide the service completely within the editor, or in part by communicating with an external program. Using an external program is preferable for many applications, in large part because editor extension languages are generally slow and specialized to text manipulation. Frequently, an external program written in another programming language can be created more easily and perform the task much more efficiently.

Another advantage of using an external program is that a single service can be shared by multiple clients using a variety of user interfaces. In addition, requirements for inter-user coordination or centralization of large databases, e.g., a netnews or a bibliographic search database, can make it desirable to centralize the service and access it with user interfaces running at other sites. For example, since a simple query can cause many database accesses, it is often most efficient to send a single message containing the query, run the query at the server, and then return the results, than it is to copy all or part of the database to the client site and then run the query there.

Communication with an external program can be provided in several ways, including files, system-supported inter-process communication (IPC), a low-level network communication protocol such as TCP, or remote procedure call. Using files for communication results in restrictive, awkward, and potentially inefficient interfaces between the editor and services. Using system-supported IPC requires the service to run on the same kind of system, yet many organizations today must cope with a heterogeneous environment, in which different machines running different operating systems are connected via a network. In addition, the use of IPC in many systems also requires the service to reside at the same site as the editor.

Using a network communication protocol such as TCP or RPC avoids the limitations imposed by the use of files or IPC for communication with an external program. A service that provides a network interface can run on any site in the system regardless of the hardware or operating system at that site, as long as that site supports the network protocol being used.

We believe that RPC is preferable to a lower-level network protocol, for the simple reason that it is easier to use and understand, is better integrated into programming languages, and allows the programmer to ignore many of the details of communication. Our experience has also been that it is relatively easy to take an existing single-site service and construct an RPC interface to it so that it can be accessed from other sites.

Given an RPC mechanism that can be used from within an editor, there is a great deal of flexibility in how services are accessed. Some services might be accessed internally by the editor. (For example, an editor being used to read mail might periodically poll a mail repository to check for new mail.) Other services (perhaps most) might be accessed as the



direct result of user action. In the latter case, there are a number of ways in which the service could be invoked. In Emacs, an action can be initiated either by typing a command directly to the Elisp interpreter, as the result of a particular key sequence, or by a menu selection. Such an action could consist of a single remote invocation, or could invoke a complex Elisp program that invokes one or more remote operations.

The arguments of a remote invocation can be obtained from several different sources. For example, if the invocation is made by an Elisp program, the arguments could be local variables in the program, or expressions that use local variables. Alternatively, the arguments could be obtained by extracting text from a selected region of a buffer. In the case of a command typed directly to the Elisp interpreter, the user can type the arguments directly as part of the command or be prompted for them by the editor. Similarly, the results of a remote invocation can be handled in several different ways. They could be stored in local variables of an Elisp program (and perhaps used as arguments of subsequent invocations), displayed in a new buffer, or used to modify an existing buffer.

We note that the discussion above applies to most, if not all, user interface systems, and not just to editors. We believe that the advantages we have cited for a dynamic extension language, for providing services by communicating with external programs, and for using RPC to communicate instead of some other mechanism, could be obtained with most user interface systems.

### 3 Mercury RPC

To support the kind of capabilities we envisioned for our distributed services, we used the Mercury system[LBG\*88]. The Mercury system was designed to support efficient and flexible communication among heterogeneous program components. Communicating modules can be written in a variety of programming languages, running on different operating system and hardware substrates. In these respects, Mercury is similar to many other RPC systems. Mercury differs from most other RPC systems in several important respects, some of which are particularly important in the context of adding RPC to Elisp:

- Mercury is designed to accommodate heterogeneity. For example, Mercury does not define a single application interface style or data representation, but allows each distinct language to define an interface that is stylistically and culturally compatible with that language. Such interfaces are called *veneers* since they provide a veneer between the language and the Mercury system. Our current prototype has veneers for C, Argus [LS83], Symbolics Common Lisp, and Elisp.
- Both synchronous and asynchronous calls are supported in a uniform manner. A server exports a collection of operations to be called by clients; each client then chooses for each call whether it should be synchronous or asynchronous. Mercury allows there to be several *streams* of requests between a single client and server; each call from the client to the server occurs on a particular stream (regardless of whether the call is synchronous or asynchronous). All calls on a single stream are delivered to the server in the order in which the invocations were made by the client. Synchronous calls are optimized to minimize latency; asynchronous calls (also called *stream calls*) are optimized to maximize throughput. Furthermore, calls may be made that return no results. Such calls, referred to as *sends*, can also be used to maximize throughput.

- Binding of clients to servers is quite dynamic. A server exports an operation by creating a *port*, which can be invoked to run a remote operation. Ports are first-class objects that can be passed as arguments and results of remote calls. A port can be made available to a client by recording it in a name server, or by passing it as an argument or result of an invocation on another port.

Accommodating heterogeneity is important for two reasons: it permits the use of the editor extension language as one of the supported languages, and it enhances the range of services that can be provided and accessed.

The ability to make asynchronous calls in Mercury, which means that the client program does not have to block waiting for the result, is important in Elisp because Elisp is a single-threaded environment: there is only one thread of control. If an Elisp program blocked while waiting for the result of an RPC, the entire editor would be blocked. Using asynchronous calls avoids this problem. As mentioned above, asynchronous calls are typically optimized to maximize throughput, but the client can force a buffered call to be sent immediately if he needs the answer quickly and does not intend to make further calls.

The dynamic nature of ports means that it is easy to provide access to a new service. If a user learns about a new service, he can easily write some simple Elisp code that will retrieve the ports exported by the service and then invoke the ports as appropriate.

Elisp code can also create ports that are passed to other sites and then called. Thus, the editor can act as a “server.” One situation in which this capability is useful is when the editor can make use of incremental results. For example, a single database query might generate a large number of results, which should be displayed incrementally to the user. This can be achieved by having the editor create a “call-back” port, which is passed to the database server as part of the query. The RPC requesting the query can return immediately, and the server can then make a series of calls to the call-back port provided in the initial RPC to deliver the results. As each result is delivered, it can be displayed to the user. In this way, the user can continue to do other work while the query is being processed, and can also abort the query after seeing some of the results. (A similar structure is supported by the “pipes and procedures” model developed by Gifford [GG88] and is used in his Walter database system [Gif85].)

## 4 Elisp Veneer

The Mercury Elisp veneer is specific to the Emacs environment, and attempts to provide the Mercury semantics in a manner that is natural and convenient for the Elisp programmer. In designing the veneer, we tried to maintain the spirit of the Lisp environment that exists within Emacs. A significant aspect of the Elisp environment is the ease with which one can generate and use new functions. Since the port is the natural analog in Mercury of the Lisp function, we tried to minimize the number and complexity of the steps required to make and use ports. In essence, there are three steps required to use remote operations from the Elisp veneer of the Mercury system. These are:

- Define the signature, i.e., the type, of the remote operation.
- Create or obtain a port for that remote operation.
- Invoke the port to execute the remote procedure.

To satisfy the needs of heterogeneous remote communication, the values of the Mercury system are more strongly typed than is typical of values in the `Elisp` environment. A Mercury port is defined with a signature that specifies the types of the input arguments, the return values, and the exceptions for the port. The syntax for this definition in the `Elisp` veneer is:

```
(hg-define-port PORTTYPE INPUTS OUTPUTS &optional EXCEPTIONS)
```

A call of this form to **hg-define-port** defines a new Mercury type, associated with the symbol `PORTTYPE`. Objects of this new type are ports that take arguments whose types are specified by the input argument list, `INPUTS`, and return results whose types are specified by the return argument list, `OUTPUTS`. (The optional argument to **hg-define-port**, `EXCEPTIONS`, is a list of exception descriptions, each of which consists of an exception name and a list describing the types of the results that can be returned when that exception is raised.) The types in these lists must be either one of the predefined scalar Mercury types, such as **hg-int32**, or a user-defined type. User-defined types are represented by symbols that have associated procedures to encode and decode the network representation of that type.

An example of a port definition is:

```
(hg-define-port testport (hg-int32 hg-int32)
                        (hg-int32)
                        (("overflow" (hg-int32))))
```

This `Elisp` expression defines a new Mercury type called **testport**. The signature for **testport** specifies that it takes two integer arguments, normally returns an integer result, but may raise the exception "overflow," which also has an associated integer result.

One of the more powerful concepts embodied in the Mercury system is the lack of distinction between clients and servers. This power is manifest in the treatment of ports as first-class values in the system. Any participant in the system is able to create and pass ports to any other participant. Thus, the editor itself could be act as a server. One situation in which this could be useful was described above: the editor could create a "call-back" port for receiving incremental results of a database query. Another example is in an interface to a mail repository. Rather than polling the repository for new mail, the editor could create a port to be used by the repository to notify the editor when mail arrives for a particular user.

In the `Elisp` veneer, it is easy to create a new port with a given signature. This is because a side effect of invoking the **hg-define-port** function, which defines the signature, is the creation of a function to create new remote functions. Use of that function registers an action routine as a new remotely invocable procedure and creates a new port of the given type that refers to that remotely invocable procedure.

For example, to create a new port of type **testport**, one need only give an action routine to the create routine, **create-testport**. Thus, if the action routine **test-action** abides by the signature for a **testport**, one can register it as a new remote procedure and create a port that refers to it as follows:

```
(fset 'test (create-testport 'test-action))
```

The call to **create-testport** creates a new identifier for a port, associates it in various system tables with the action routine **test-action** so that incoming calls to the port can

be directed to the appropriate place, and returns the new port that refers to the action routine **test-action**. The assignment above saves the newly created port in the variable **test**.

Note that in the example above we have done an **fset** of the port value generated by the **create-testport** function<sup>2</sup>. This is done to preserve as much as possible the analog between ports and Lisp functions. In fact, the representation of a port in Elisp is very similar to that of function, so that invoking a port is identical to invoking a function. Thus, the simple expression:

```
(test 1 2)
```

results in a remote invocation of the port named **test**, which in turn results in the invocation of the action routine **test-action**.

The preceding example is rather trivial, since the invoker and the action routine are co-resident in the same editor session, and because it is a relatively simple invocation. In general, ports in Mercury are quite powerful: they provide for a wide variety of data types, multiple return values, and exceptional returns with associated values. In Elisp, these capabilities are supported by allowing for general encode and decode of arguments, for returning a list of values to accomodate multiple return values, and by generating Elisp **signals** corresponding to the Mercury exceptions.

#### 4.1 Invoking Ports

Mercury ports have a dual nature: they must be able to be treated as functions, for making remote invocations; and they must be able to be treated as values that identify the remote operations, for use when passing a port as an argument or result.

Port invocation is essentially a request-response exchange between client and server. Thus, when treated as a function, a port must identify the appropriate remote operation, take the appropriate arguments, encode them into a network message, and send a message to the remote module that provides the operation referred to by the port. When the invocation completes, a response message is returned to the client; the client must accept the message, decode the results, and return them to the user program. Associated with each port in Elisp is a unique identifier, which allows the communication subsystem to find the module that provides the operation referred to by the port. Also associated with each port is a stub routine that handles encoding and decoding of arguments and results, as well as message transmission. This is analagous to standard RPC stub procedures.

To reduce the volume of code dedicated to port stubs, the Elisp veneer uses a layered approach to stub design. All stubs are supported by two generic stub routines: one at the client and another at the server. On the client side, the generic stub routine takes several arguments: the unique identifier for the port, the actual arguments of the remote call, and a template describing the types of the input and return arguments and the exceptions that may be raised. Thus, all encoding and decoding of argument and result values can be done in a table-driven fashion. A similar strategy is used on the server side.

The representation for a port is just a function that calls the generic stub, passing it the appropriate arguments. This function has the following form:

```
(lambda (&rest args)
```

---

<sup>2</sup>Fset is the Lisp function that is used to assign a function value to a variable.



```
(generic-stub unique-id
               input-types
               return-types
               exception-types
               args) )
```

Thus, an invocation of a port results in an invocation of the generic stub routine, passing the port identifier, the type descriptions, and the list of actual arguments. The type descriptions are generated by **hg-define-port** when a new port type is defined; this information is the same for all ports of the same type. Ports of a given type are created either by the creation operation associated with the type, or by the decode routine for the type. (The decode routine is used to receive ports from other modules; it is described in the next section.) Whenever a port is created, the unique identifier for that port is combined with the type information associated with the port type.

For example, the invocation

```
(test 1 2)
```

calls the function assigned to **test**; this function calls the generic stub, passing it the port identifier created when **test** was created, the type lists (**hg-int32 hg-int32**), (**hg-int32**), and (**"overflow" (hg-int32)**), and the list of arguments (**1 2**). The generic stub encodes the arguments using the encode routines associated with **hg-int32**, sends the invocation message, receives the reply, and decodes the result appropriately.

Stream calls can be performed using a mechanism similar to *promises* [LS88], which were designed for using stream calls in Argus.

## 4.2 Obtaining Ports

As illustrated above, ports can be obtained simply by using the creation functions created by **hg-define-port**. However, creating ports and just using them locally is not very useful. To use remote services, their ports must be obtained. This is simple to do in Mercury, since ports are first-class values: they can be transmitted as arguments and results of remote invocations. (In fact, arbitrary structures containing ports, such as records, can be transmitted.) Thus, remote invocations can be used to distribute ports to other modules and to obtain their ports.

A port is sent as an argument or result of a remote invocation by transmitting the port's unique identifier. To implement port transmission in a veneer, one needs routines to encode and decode port values of each port type. In the Elisp veneer, these encode and decode routines are generated automatically by **hg-define-port** and associated with the symbol naming the porttype being defined. Thus, after defining a porttype, one can easily define and use another port that passes a port of that type.

Passing ports as arguments and results of remote invocations makes it easy to obtain new ports, but does not solve the problem of obtaining the first remote port. To get started, a client must first talk to a globally known service. Mercury provides a *catalog* that every Mercury module knows how to talk to. The catalog can be used to store ports, which can then be retrieved by other modules. Thus, a service might put its ports in the catalog, and an Elisp program might retrieve the service's ports from the catalog during an editing session. The current Mercury catalog is rather primitive, simply mapping string



names to individual ports. Nonetheless, if one can access the catalog, one can then get to arbitrary remote services, as long as they have stored their ports in the catalog (or there are ports in the catalog that will return their ports, and so on).

Accessing a new service is thus quite straightforward. The types of the ports provided by the service must be defined with **hg-define-port**. Then, the service's ports must be obtained, typically from the Mercury catalog. Thus, new services are easy to add dynamically to an Emacs environment. In addition, the editor itself can easily define its own ports and make them available to other modules.

The Elisp veneer also provides a number of other operations on ports. These operations include functions to control whether an invocation should expect a reply or not, how the invocation should be sequenced and whether the requests and responses should be buffered.

Finally, we note that Mercury also provides a mechanism for documenting remote operations. Given a port one can inquire about the documentation of that port; the result of the inquiry is a string, which is provided by the port's creator. Access to the Mercury documentation is uniformly integrated into Elisp, by allowing users to use the standard Emacs help facility to find documentation on a port. Similarly, when a new port is created, the port's documentation is picked up automatically from the documentation of the action routine being associated with the port.

## 5 Performance

An initial version of the GNU Elisp veneer is operational. Most of the code added to GNU Emacs was obtained from the previously constructed Mercury C veneer. In addition, three modules of C code were added and two existing modules in the GNU Emacs implementation were modified.

The results of performance measurements for the GNU Elisp veneer are described below. All these measurements have been made between MicroVAX<sup>3</sup>-IIs running Ultrix<sup>4</sup> 3.1, connected by an otherwise idle Ethernet.<sup>5</sup> Each test was performed using a loop that repeatedly executed the expression being tested. The reported results are the expression execution time with the loop overhead subtracted out.

The initial tests were designed to show the performance of local Elisp operations. There are two classes of functions in Elisp: built-in functions, written in C, and native Elisp functions, written in Elisp. Very simple functions of each class were tested, including the built-in function **identity**, which does nothing but return its input argument, and a user-defined null function that does nothing but return **nil**. Table 1 shows the results of these measurements. As can be easily seen, the native Elisp performance is not particularly fast. Fortunately, since most Elisp operations are simple, and intended for interactions with human operators, tremendous speed is not necessary.

There is another variant for user-defined functions: user-defined functions may be *byte-compiled* into a more efficient form for faster loading. When byte-compiled, however, the **local-null** function ran more slowly than when not byte-compiled.

Table 1 also shows the times for an operation with a single integer input argument and one with a single integer return value. The added cost for the input argument was 0.5 ms.,

<sup>3</sup>MicroVAX is a trademark of Digital Equipment Corporation

<sup>4</sup>Ultrix is a trademark of Digital Equipment Corporation

<sup>5</sup>Ethernet is a trademark of Xerox Corporation

Function	Time
(identity nil)	0.6 ms.
(local-null)	1.2 ms.
(local-int-in 1)	1.7 ms.
(local-int-out)	1.2 ms.

Table 1: Local Function Execution Times

Function	Time
(remote-null)	42 ms.
(remote-null-send)	19 ms.
(remote-int-in 1)	43 ms.
(remote-int-out)	43 ms.
(remote-string-in s1) ; Length=1	44 ms.
(remote-string-in s100) ; Length=100	50 ms.
(remote-string-in s500) ; Length=500	73 ms.

Table 2: Elisp to C Remote Function Execution Times

while the return value cost nothing extra.

Performance results for remote calls from Elisp to C are shown in Table 2. A set of remote functions analogous to the set of local functions were tested. An additional test for a **send** to the server is also shown. This test shows that a significant performance improvement, better than a factor of 2, can be achieved when sends are used instead of blocking calls. Some tests for functions with an input string argument are also shown. These show the incremental cost of a byte of string data to be about 60 usec per byte. This high time is a consequence of byte-by-byte encoding and decoding of bytes of the string. Encode and decode operations that batched bytes together could improve this figure significantly. For comparison, the performance of a C client using an Elisp server providing a null function is shown in Table 3.

Overall, the costs for Elisp remote calls are significantly greater than for local calls. In the current Mercury implementation, the time for a null RPC between a C client and a C server is less than 18 ms. Since the Elisp communication subsystem implementation is identical to the implementation used for the C veneer, it is clear that most of the performance loss is in the higher levels of the Elisp veneer. Much work has been done on achieving high performance RPC (e.g., [Che86]), and significant opportunities for optimization of the Elisp veneer still exist. (In fact, almost no performance tuning has been done for any of the Mercury veneers at this point.) For example, making a remote invocation in Elisp involves a number of Elisp function calls; since each of these involves significant overhead, reducing the number of function calls, for example by substituting the bodies of some functions in-line for the calls, could make a big difference. Implementing more of the veneer directly in C, rather than in Elisp itself, could also help. In addition,

Function	Time
(remote-null)	41 ms.

Table 3: C to Elisp Remote Function Execution Times

the lower levels of the veneer were originally designed for C, so they make assumptions about data representations that are appropriate for C, but are less than ideal for Elisp; redesign to make the pieces more compatible would also improve performance.

Even so, the performance is similar to that achieved for the echo of a null line to the UNIX `cat` program using IPC, which was measured to be 27 ms. Furthermore, the ratio of local to remote costs is similar to the ratio seen in other RPC environments. Finally, one should note that the objective of providing remote services is often not to do a large number of small calls, but to do a significant task in response to a single call. In such cases the cost of the RPC can be dominated by the service time itself. In fact, from our experience so far, the performance has already been acceptable for use in a number of services.

## 6 Experience

In our work to date with the Elisp veneer, we have integrated access to several applications. These applications have given us some experience with the use of an editor as a front-end to a number of distributed services written in different languages.

One of the simplest applications allows an Emacs session to be a client of the Mercury catalog. The Mercury catalog has implementations in C, Argus, and Lisp. As discussed earlier, it provides a simple mapping of names to ports, allowing a service to store its ports for later retrieval by potential clients. The catalog is the primary means by which a Mercury client can locate other services. An Emacs session can retrieve ports of services from the catalog, either directly with a user-issued command, or with an RPC issued by an Elisp program.

An interesting question is whether one could write a single “generic” piece of Elisp code to be used to integrate new services, so that users could browse through the ports exported by services in the catalog and then invoke them without writing any additional Elisp code. For ports whose arguments and results are restricted to a small set of data types (e.g., strings and integers), this should not be too difficult, but for more general types of arguments and results it is less clear how it could be done.

Some other simple services include an Elisp client interface to a simple database service, an inter-emacs “talk” facility, and a personal mail nicknaming facility, which is to be integrated into a more general user naming facility. A more sophisticated database client for access to a full Webster’s dictionary is under development. This application will look up the definition of a word (e.g., the word under the editor’s cursor), and also check the word’s spelling and provide possible corrections. These applications have demonstrated the ability to access heterogeneous services, and have shown how easy it can be to integrate access to a service into Emacs.

As an example of using the editor as a server rather than as a client of other services, we have implemented a simple remote editing facility. When the editor starts up, it creates a port and stores it in the catalog. Other programs can retrieve and call the port, passing it text to be edited. The text is then displayed in a new buffer in the already running editor. When the user indicates that he is done editing the text, the port invocation returns, passing the modified text as a result. Such functionality could be used in programs such as Latex, which frequently require user interaction to correct erroneous text. Using a remote editing facility in this way would allow the user to use

more powerful editing capabilities than provided by Latex, and would also allow him to use his customized editing environment. (The Latex source would probably need to be modified to do this, since it would need to call the editor.) A remote editing facility could also be used to edit information from environments in which the user's preferred editor does not run.

A shared buffer facility has also been implemented. This facility allows multiple users to simultaneously edit a single shared buffer in which each sees the others' updates. Simple locking primitives are provided that allow a user updating a region of text to prevent updates by other users to the same region. Such a mechanism could be a step toward more sophisticated facilities for computer-supported cooperative work. One interesting point about the implementation of the shared buffer facility is that stream calls are needed to achieve adequate performance; if blocking RPCs are used to send updates from one editor to another, the delays seen by the user become intolerable.

Experience has also been gained with two larger applications. First, the PCMAIL repository [CL86] was converted to run under Mercury, and an Elisp client interface to the PCMAIL facility was implemented. PCMAIL is an enhancement of RMAIL (the standard GNU mail interface). PCMAIL uses the RMAIL user interface, but replaces the local mail system with a mail repository accessible over the network. The mail repository is implemented in C, and provides robustness and integrity guarantees that are not normally provided with standard UNIX mail. In the original implementation of PCMAIL, access to the mail repository was provided via an application-specific protocol, called DMSP, that is layered on a data representation protocol that is itself layered on TCP. As with RMAIL itself, the basic editor extension language turned out to be inadequate in building PCMAIL. In this case, the data manipulations required for DMSP were too complex to be done solely in Elisp and still provide adequate performance. To solve this problem, the capabilities of Elisp were augmented by adding DMSP-specific code libraries into the editor source. In contrast, once Mercury RPC had been added to Elisp, no further modifications of the Emacs source were needed; all the user interface code was written in Elisp.

We note that the PCMAIL user interface in the editor has all the usual advantages of using an editor for mail, including the ease of viewing received messages, extracting text from received messages, and composing mail messages. Separating the mail repository from the user interface allows different interfaces to be used by different users, while the integrity and reliability guarantees need to be implemented only once in the repository itself. In addition, the implementation of PCMAIL based on Mercury RPC is interesting for several reasons. First, it involves communication between programs written in different programming languages, since the server is written in C and the user interface is in Elisp. Second, converting PCMAIL to use Mercury instead of DMSP demonstrated the need for a simple mechanism for customizing encodes and decodes. For efficiency, it was important to decode received messages directly onto the end of an editor buffer, rather than converting them to some internal representation of a message as a collection of strings. The representation for Mercury data types (both built-in and user-defined) was chosen so that customized encode and decode routines would be easy to incorporate in the stub for a port.

The second large application involves the Walter database retrieval system. Walter is a database system used to access articles from the New York Times and Associated Press news wire services [Gif85]. The existence of a large database and the need for sophisticated query processing resulted naturally in its design as a distributed service, with a powerful



back-end providing most of the data storage and computation, and separate front-ends acting as user interfaces. Walter was initially implemented using the pipes and procedures model for communication [GG88]; it has been converted to run under Mercury.

The Walter system illustrates some of the advantages of using an existing editor as the user interface to a service. The basic functionality required by Walter users is making queries (represented as lines of text) and displaying responses, either various summaries or whole articles. The original user interface provided this functionality by behaving as a very simple editor with windows for queries and their responses. It did not, however, provide an integrated environment in which the usual editor capabilities, such as cut and paste, could be used on the data retrieved. Thus, using an existing editor as an interface not only avoids much of the effort of creating a new user interface, but can also increase functionality by permitting use of standard editor facilities. A prototype version of the Walter system using Mercury and the Elisp veneer is operational.

## 7 Conclusion

Adding RPC to Elisp has greatly enhanced the power of the editor, by allowing it to be used to access a wide variety of services, which can be distributed over a number of machines with varying hardware and operating systems and can be written in different programming languages. An RPC interface for a service can typically be constructed very quickly. A simple user interface for a service can then be provided quickly by writing a small amount of Elisp code. In addition, the uniformity and integration achieved by accessing a variety of services from a single editor provide further benefits, while at the same time allowing other customized front-ends to continue to access the service.

Other network communication protocols, such as TCP, have been added to editors in the past, and have provided similar benefits. However, RPC is superior to lower-level protocols because it is easier to use: it is easier to construct an RPC interface for a service, and it is easier to write the Elisp code that accesses the service. As outlined above, RPC is also superior to other methods for communicating with an external service, and using an external service is generally superior to implementing the service completely within the editor.

At present, the Elisp veneer provides a substantial portion of the Mercury functionality, including stream calls. Several services are operational, and the system shows promise as a useful integrated interface for a number of services. Already, work is in progress on the dictionary access service. More applications, such as bibliographic searches and phone books, should be generated in the future. Further efforts are still needed on the Elisp veneer to improve performance.

Overall, we believe that using RPC for service access from an common integrated interface is a powerful and useful concept. Using a text-only editor as an interface has already shown promise, but expanding the scope of the interface is also possible. For example, one could also provide this kind of service interface in multi-media editors. One could even imagine having active components of a document that use remote invocation to call programs that dynamically generate a portion of the document. Alternatively, one could expand the scope of the service interface to be a window system. In this way, local services such as cut and paste, which are available globally in some existing window systems (such as the Macintosh or the Symbolics Lisp Machine environment), could be



expanded to provide access to powerful distributed, heterogeneous services.

## 8 Acknowledgments

The authors would especially like to thank Mark Lambert, who was one of the principal implementors of the *Elisp veneer*, Staffan Blau, who was a major contributor to the *C veneer* implementation on which the *Elisp veneer* relied, Rob Austein and John Wroclawski, who designed and implemented stream calls in the *Elisp veneer*, and Rob Cote, who implemented the *Mercury* version of *Walter*. In addition, the authors would like to thank the other members of the *Mercury* research team, especially Toby Bloom, Dorothy Curtis, Dave Gifford, Barbara Liskov, Bob Scheifler, Liuba Shrira, and Karen Sollins, for their work on *Mercury* in general and on the *Elisp veneer* in particular.

## References

- [BN84] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [Che86] D. Cheriton. VMTP: a transport protocol for the next generation of communication systems. In *Communications Architectures and Protocols*, pages 406–415, August 1986.
- [CL86] D.D. Clark and M.L. Lambert. *PCMAIL: A Distributed Mail System for Personal Computers*. Request for Comments RFC 993, Defense Advanced Research Projects Agency, December 1986.
- [Dig85] Digital Equipment Corporation. *VAX language-sensitive editor user's guide*. Maynard, MA, 1985. Part Number AADB33ATE.
- [GG88] D. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3), August 1988.
- [Gif85] D. Gifford et al. An architecture for large scale information systems. In *Proceedings 10th ACM Symposium on Operating Systems*, pages 161–170, 1985.
- [Lan86] K. Lantz. On user interface reference models. *SIGCHI Bulletin*, 18(2), October 1986.
- [LBG\*88] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the *Mercury* system. In *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, January 1988. Available as MIT LCS Programming Methodology Group Memo 59.
- [LS83] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.
- [LS88] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM*

*SIGPLAN '88 Conference on Programming Languages Design and Implementation*, June 1988.

- [Sta81] R.M. Stallman. *EMACS: the extensible, customizable self-documenting display editor*. Technical Report, MIT, 1981.
- [Sta86] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, MA, 1986.
- [Wat86] R.C. Waters. *KBEmacs: A Step Toward a Programmers Apprentice*. Technical Report MIT-AI TR-753, MIT, 1986.

Joel S. Emer  
DEC

William E. Weihl  
M.I.T.

Joel S. Emer is a consulting engineer with the Distributed Systems Architecture and Advanced Development group of Digital Equipment Corporation. Since joining Digital in 1979, he has worked on VAX processor and memory subsystem architecture and performance, and systems performance analysis. He also led the distributed systems research team whose work resulted in the DFS product. Dr. Emer recently finished work on a research project at MIT investigating issues in heterogeneous distributed computing. Before coming to Digital, he received his Ph.D. (1979) in Electrical Engineering from the University of Illinois at Urbana-Champaign. He also earned B.S. (1974) and M.S. (1975) degrees in Electrical Engineering from Purdue University. Dr. Emer is a member of Tau Beta Pi, Eta Kappa Nu, ACM, and IEEE.

William E. Weihl is an Associate Professor of Computer Science and Engineering at M.I.T. His current research interest focus on parallel and distributed computing, particularly in the areas of programming methodology, programming languages, specification techniques, synchronization, and fault-tolerance. Before joining the faculty at M.I.T., he received the S.B. degree in Mathematics in 1979, the S.B. and S.M. degrees in Computer Science in 1980, and the Ph.D. degree in Computer Science in 1984, all from M.I.T. Dr. Weihl was one of the principal designers of the Argus programming language and system, which is designed to support the construction of reliable distributed systems, and the Mercury communication system, which is designed to support efficient, flexible communication in heterogeneous distributed systems. He has also made significant contributions to the theory of atomic transactions, particularly in the design of highly concurrent type-specific concurrency control and recovery algorithms.



# The UNIX® System Math Library, a Status Report

*Joel D. Silverstein, jds@attunix.att.com*

*Steven E. Sommars, sesv@iwtsf.att.com*

*Yio-Chian Tao, tao@iwtsf.att.com*

*AT&T Bell Laboratories, Summit, NJ & Naperville, IL*

## ABSTRACT

Changes have been made to the UNIX® System math library (libm) to improve accuracy and to conform to standards. Several improvements were made for C/SVR4, most notably in the trigonometric functions, `fmod(x,y)`, `lgamma(x)`, `pow(x,y)`, and `erfc(x)`. Significant improvements come from use of some 4.3BSD functions. While additional algorithmic work is needed to improve accuracy and performance of some functions, a more important task is to standardize interfaces including exception handling.

### 1. Introduction

While not an early strong point, floating point computations are now extensively used on computers running the UNIX® operating system. The accuracy of the math library component of those computations has been a concern for some time, but until recently little attention has been paid to improvements. It is widely known<sup>1</sup> that the UNIX system math library on many implementations is inaccurate. We wanted to measure and improve the accuracy of libm functions in the latest AT&T C compiler, C/SVR4<sup>2</sup>. We developed a test suite for measuring the accuracy of all libm functions. Motivated by these measurements, many functions were modified or replaced (often by the Berkeley 4.3BSD version).

### 2. Desirable properties of a math library

The desirable, but sometimes contradictory, properties of a math library as seen by the library user include:

- *speed*. The run-time performance of a math function is one of its most visible properties.
- *size*. Different algorithms require varying amounts of system ROM/RAM, possibly a limited resource.
- *accuracy*. It is not generally possible to guarantee the same accuracy for the transcendental functions that one obtains with addition, multiplication, etc. Obtaining results bearing some resemblance to the mathematically exact value should be important to users. Values at a few special points (e.g., `sin(0.)`, `pow(2.,2.)`) should be exactly what the user expects. The IEEE floating point standard ([IEEE754]) specifies the accuracy of some

---

1. Spafford and Flaspohler (see [Spaf]) ran the Cody/Waite tests on a variety of machines running the UNIX operating system. While AT&T computers fared well on the whole, some implementations returned results with few accurate digits.

2. A C compiler supporting ANSI C, often distributed with System V Release 4.0. The exact product name varies among implementations. Comparisons are made to CPLUS 4, the compiler distributed with SVR3.

operations (e.g., add, subtract, multiply, divide, remainder) but places no requirements upon others (e.g.,  $\sin(x)$ ,  $\exp(x)$ ). It is difficult to define and implement a single accuracy threshold appropriate for all applications.

- *monotonicity*. If a function  $f(x)$  is monotonically increasing over an interval, then for any  $x \leq y$  chosen from that interval,  $f(x) \leq f(y)$ . Monotonicity failures can cause problems, for example, in evaluating divided differences. The only function we have observed to violate monotonicity for C/SVR4 is  $\lgamma$ . We would not be surprised to learn of others.
- *range limits*. A function whose range is bounded should not go outside those bounds. If  $\sin(x)$  returns a value greater than one, the programmer who evaluates  $\text{sqrt}(1 - \sin(x) * \sin(x))$  will be unpleasantly surprised.
- *symmetry*. The math library's approximations for symmetric/antisymmetric functions should preserve<sup>3</sup> those properties.  $\sin(x) = -\sin(-x)$  should hold for all finite values of  $x$ .
- *reproducibility*. While it may be desirable to have exact reproducibility (same argument gives same result) for math functions across implementations, in practice this would severely constrain implementations by forcing them to use identical approximations. CORDIC approximations (see [VOLD]) may be fast and accurate when suitable hardware support is present, but are not good techniques otherwise.
- *support of standard interfaces*. The interface to the math library (e.g., language binding) should meet the user's expectations.
- *exception mechanism*. The behavior of a function when presented with an argument outside the normal domain (e.g., at a singularity) should be well-defined and user configurable.
- *transportability of binaries*. It is desirable for a compiled program to execute on a wide set of target machines (e.g., on all systems with the same instruction set). Direct use of hardware floating point and transcendental approximations for best performance may preclude transportability to systems lacking such hardware. This has led to floating point compilation switches/modes that let users choose varying degrees of portability and performance.

### 3. Portability & libm

The UNIX operating system runs on computers supporting a wide variety of floating point formats. While the interfaces to the math libraries are quite general, it is not feasible for a single libm implementation to do a good job on *all* computers. Because it is standardized and widely used by many computer vendors, our efforts were concentrated on IEEE754 [IEEE754] floating point, primarily double precision (53 bits significand, 64 bits total size).

Even within IEEE754, there are many variations in hardware. Several floating point processors (e.g., AT&T 32206, Intel 80387, Motorola 68881) have approximations for some transcendental functions built-in. Some systems (e.g., 32206) efficiently support double-extended ( $\geq 64$  bits of precision) floating point. If these facilities are available, the libm developer should feel free to use them<sup>4</sup>. But they are not universal. Floating point hardware is

3. The symmetry property may not be applicable in some rounding modes.

4. For example, when a WE@32106 is known to be present, the  $\exp(x)$  and  $\text{atan}(x)$  functions deliver accuracy near 0.5 ulps by using extended precision.



an extra cost option on some systems. This paper will concentrate on 53 bit arithmetic with no extensions. The results discussed here should be achievable on any IEEE754 system.

Even within the domain of IEEE double precision computers, there are many algorithms to choose from. Tradeoffs between size, speed, and accuracy are made. The computed values of transcendentals will vary among implementations, and sometimes even from run to run of a single binary if the math function is part of a changeable shared library.

#### 4. Accuracy Metric

An ulp (Unit in the Last Place) is a floating point accuracy metric<sup>5</sup>. For IEEE754 double precision, we define<sup>6</sup> one ulp as  $ulp(x) = 2^{\log_b(x)-52}$  where  $\log_b(x)$ , one of the recommended IEEE754 functions, is the unbiased exponent of  $x$ . The error in ulps is calculated from the approximate ( $F(x)$ , a double precision floating point number) and the exact result ( $f(x)$ , generally a transcendental number).

$$\text{error(in ulps)} = (F(x) - f(x)) / ulp(f(x)).$$

To illustrate the usage of ulps, assume that the nearest double precision approximation for  $\sqrt{x}$  differs from the exact value of  $\sqrt{x}$  by 0.221 ulps (rounded to 3 places). The only possible errors from measuring the accuracy of a particular implementation are  $0.221 \pm N$ ,  $N$  integer. Any implementation that returns a value with an error of 0.221 ulps is as accurate as possible in round-to-nearest mode. The 0.221 ulp error is inherent in using finite precision floating point. In IEEE round-to-nearest mode, if the error for a function is always in the range  $[-0.5, 0.5]$  ulps, then we should be satisfied; no improvement is possible. This is the situation for  $\sqrt{x}$  in C/SVR4. An important tool for presenting and interpreting error results is the "ulp plot," a scatterplot of the ulp error versus argument over a given domain. Error trends and biases are more easily seen graphically. If the  $|ulp \text{ error}|$  is less than 0.5 ulps, the ulp plot is usually uninteresting<sup>7</sup>.

A warning. Since ulps are a measure of relative error, it is only appropriate to compare directly the ulp accuracy of functions with the same precision. A single precision  $\sin(x)$  function with a maximum error of 0.6 ulps is not more accurate than a double precision  $\sin(x)$  function with an error of 2.3 ulps. For IEEE floating point, the former would have about 23 accurate bits while the latter would have about 51 accurate bits.

Another warning. For some non-elementary functions, ulps are not good accuracy metrics.

5. The M.R.E. (maximum relative error) metric defined in [Cody-Waite] is related to the ulp metric; both measure relative error. The M.R.E. is easier to compute, but is somewhat harder to interpret.

6. When  $x$  is a power of the radix, alternate definitions of an ulp have been used.

7. See, however, the discussion of random number generation for a counter example.

See the section on `j0(x)`.

## 5. Measuring Accuracy

### 5.1 Methods

Two methods are used to generate reference/accurate values for functions: run-time computation and precomputed tables. Run-time tests for `atan(x)` and `log(x)` were done to better than 0.1 ulps accuracy using Alex Liu's (UC-Berkeley) elementary function test package. All other functions were tested using precomputed tables. These were created using both `bc(1)` scripts and Brent's MP arbitrary precision floating point package [Brent]. The disadvantages of precomputed tables include the requirement for large amounts of disk storage space, the need to spend a significant amount of (one-time) CPU time in the creation of the tables, and the fact that the tables are applicable only to a single floating point format. An advantage is that they are quite accurate<sup>8</sup>.

### 5.2 Random number generation

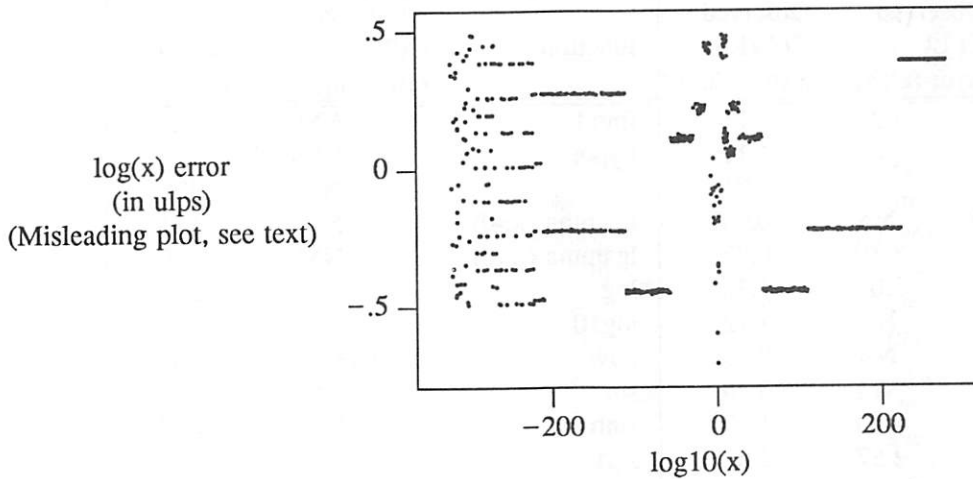
Selecting the arguments for checking each function's accuracy presented some difficulties. We originally used the `drand48()` linear congruence pseudo-random number generator. The double precision numbers generated by `drand48()` contained 48 random bits in the significand; the least significant bits were zero! This is not necessarily a source of bias, but it concerned us. We now use a modification suggested by C.S.Roberts. `drand48()` was called *twice* and the most significant bits from each call combined to create a pseudo-random floating point number<sup>9</sup>. While the test points themselves were different when we changed from simply using `drand48()` directly to using Roberts' method, there was no significant difference in the resultant ulp plots.

Another problem surfaced when measuring the accuracy of `log(x)` using a logarithmically distributed set of test points. These were originally generated over an interval `[a,b]` as suggested by Cody&Waite:

$$X = a e^{y \log \frac{b}{a}} \quad y \text{ random, uniform in } [0,1]$$

The `log(x)` results for `[a,b]=[MINDOUBLE,MAXDOUBLE]` were:

- 
8. The tables were designed to be accurate to better than 0.001 ulps. This was checked by running the full tables through both `bc` and `MP` and by running a subset through the Maple symbolic math package. All methods agreed to within 0.001 ulps.
  9. On systems supporting the `rand()` or `random()` generators, a similar modification may be used. One should be cautious about correlations between successive pseudo-random numbers.



The unexpected structure (clustering should not happen) indicates a problem in the choice of test points. Using  $\log(x)$  and  $\exp(x)$  to generate the points caused a biased set to be chosen<sup>10</sup>. Also, using  $\log(x)$  to generate test points for itself is inherently disturbing. A different procedure using Monte Carlo techniques was developed for generating logarithmic distributions. As others have observed, it is risky to place blind trust in pseudo-random number generator. Whenever we obtained unexpected results, we considered the random number generator a possible culprit.

### 5.3 Testing Domains

Each function was tested both over its full domain and over selected intervals. Errors in range reduction may be manifested separately from approximation errors within primary intervals. The number of points tested varied, but was at least 10,000 for each function.

### 5.4 Trigonometric Range Reduction

Different techniques of trigonometric range reduction have been used in math libraries and are discussed in the appendix. All functions (trig and inverse trig) should use the same trig range reduction definition. It is important for the *tests* to use the same range reduction definition as the *implementation*. The accuracy results reported here are with respect to true  $\pi$ .

## 6. Results of accuracy tests

The following table summarizes the maximum observed error (over all points tested) for the C/SVR4 math library using IEEE754 double precision arithmetic<sup>11</sup>. One cannot interpret these values as a worst case limit, but we would be surprised to see significantly larger errors.

10.  $\exp(x)$  gives a sparse set of output values. The points generated are also sensitive to the method chosen to evaluate the above formula. Overflow/underflow are possible unless care is taken. The multiplications may cause rounding errors.

11. Some routines were also implemented in WE32106/WE32206 assembly language to provide greater accuracy and/or performance. Those results are not presented in this paper.

function	Observed CPLU 4 error (ulps)	Observed C/SVR4 error (ulps)	function	Observed CPLU 4 error (ulps)	Observed C/SVR4 error (ulps)
acos	1.25	1.15	fmod	NM	0.00
acosh	NA	1.11	hypot	2.12	0.95
asin	2.27	2.01	j0	NM	NM
asinh	NA	0.90	lgamma (x>0)	NM	3.48
atan	3.30	0.82	lgamma (x<0)	NM	NM
atan2	3.03	1.32	log	1.94	0.74
atanh	NA	1.72	log10	2.41	1.61
cbrt	NA	0.66	pow	993.81	98.48
cos	NM	0.76	sin	NM	0.82
cosh	1.60	1.19	sinh	1.45	1.05
erf	2.57	2.32	sqrt	0.75	0.50
erfc	$\approx 4 \cdot 10^9$	6.64	tan	NM	1.91
exp	1.49	0.75	tanh	1.23	1.23

NA=not applicable. Function did not exist in library.

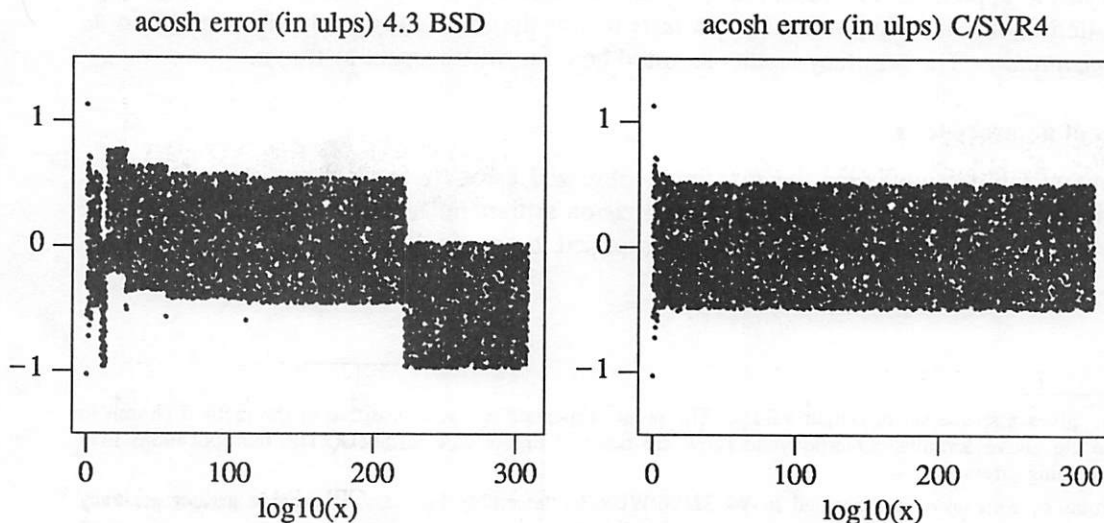
NM=not meaningful. Either some part of the function's definition changed or the ulp error is too large to be meaningful. See the text for more information.

Because of space limitations, only a few ulps plots are shown here. They do not always include the points with maximum error.

### 6.1 acos

The arccosine ( $\text{acos}(x)$ ) function showed improved accuracy as a consequence of a more accurate  $\text{sqrt}(x)$  function. While the overall error in  $\text{acos}(x)$  and  $\text{asin}(x)$  is comparable for C/SVR4 and 4.3BSD, it can be argued that the 4.3BSD implementation has a more uniform error distribution.

### 6.2 acosh



The inverse hyperbolic cosine ( $\text{acosh}(x)$ ) function (from 4.3BSD) is new for C/SVR4. A bias in the error for large  $x$  starting near  $2.28 \cdot 10^{222}$  can be seen in the original 4.3 routine. Why is this? For large  $x$ ,

$$\operatorname{acosh}(x) \sim \log(2x)$$

Since the  $2*x$  term may overflow for very large  $x$ , the 4.3BSD library uses

$$\operatorname{acosh}(x) \sim (\log(x) + \ln 2\text{lo}) + \ln 2\text{hi}$$

where  $\ln 2\text{hi} + \ln 2\text{lo}$  is  $\log(2)$  stored in two double precision words. Near  $10^{222}$  the *normalized* value of  $\log(2)$  that must be added to  $\log(x)$  is

$$\begin{aligned} \log(2) &= 12193974156572.967... * 2^{-44} & x < 2.28 * 10^{222} \\ \log(2) &= 6096987078286.483... * 2^{-43} & 2.28 * 10^{222} < x \end{aligned}$$

Just below  $2.28 * 10^{222}$  the addition loses less than 0.04 ulps to roundoff while above that point, nearly 0.5 ulp is lost to roundoff. This is the origin of the bias for large  $x$ . The C/SVR4 library was revised for large  $x$  to compute:

$$\operatorname{acosh}(x) = \log(2*x) \text{ if } x < \text{MAXDOUBLE}/2$$

or

$$\operatorname{acosh}(x) = \log(x/2) + \ln(4) \text{ if } x \geq \text{MAXDOUBLE}/2$$

Just below  $\text{MAXDOUBLE}/2$ ,  $\operatorname{acosh}$  retains the full accuracy of  $\log$ . Above  $\text{MAXDOUBLE}/2$ , the addition of  $\ln(4)$  increases the error by less than 0.04 ulps.

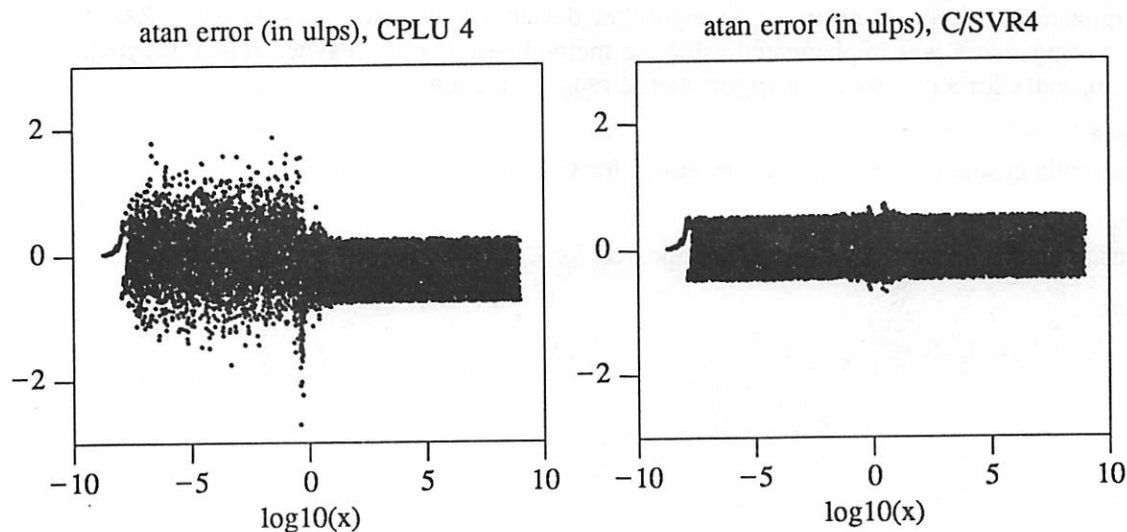
### 6.3 asin

The arcsine ( $\operatorname{asin}(x)$ ) function showed improved accuracy as a consequence of a more accurate  $\operatorname{sqrt}(x)$  function.

### 6.4 asinh

The inverse hyperbolic sine ( $\operatorname{asinh}(x)$ ) function (from 4.3BSD) is new for C/SVR4. See the section on  $\operatorname{acosh}(x)$  for a discussion of large  $x$  behavior.

### 6.5 atan



The 4.3BSD  $\operatorname{atan}(x)$  function showed improved accuracy over the CPLU 4 version. Since the trig functions use true  $\pi$  for range reduction, it was important to use coefficients for the  $\operatorname{atan}(x)$  approximation that match.



### 6.6 *atan2*

The arctangent function of two arguments (*atan2*(*y*, *x*)) is more accurate as a consequence of the improved *atan*(*x*) function.

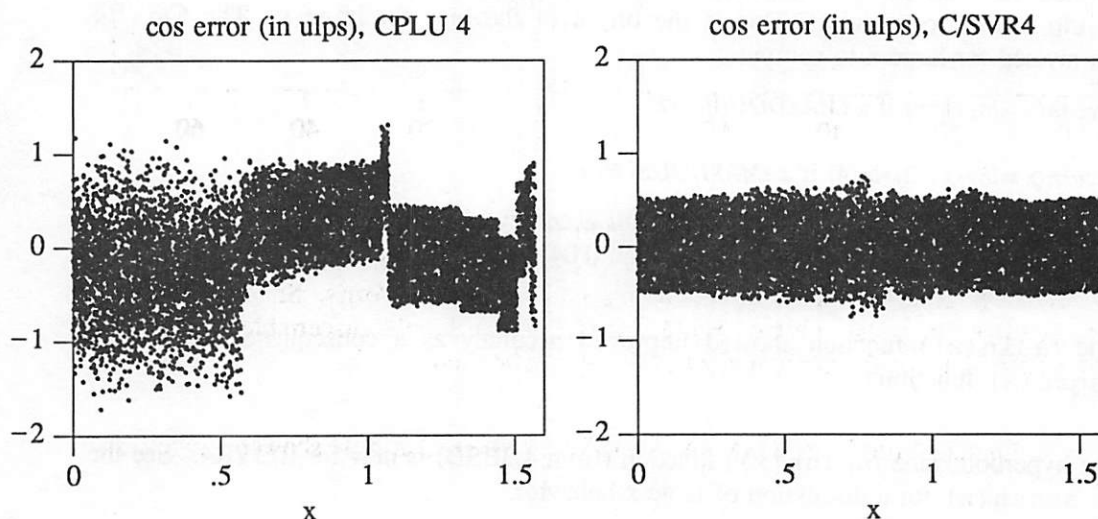
### 6.7 *atanh*

The inverse hyperbolic tangent (*atanh*(*x*)) function (from 4.3BSD) is new for C/SVR4.

### 6.8 *cbrt*

The cube root (*cbrt*(*x*)) function (from 4.3BSD) is new for C/SVR4.

### 6.9 *cos*



The previous math library implementation for cosine (*cos*(*x*)) had noticeable biases. These were eliminated in C/SVR4 by using an algorithm developed by Peter Tang[Tang1]. Range reduction using true  $\pi$  was implemented using the method described by Payne&Hanek [Payne]. See the appendix for a discussion of trigonometric range reduction.

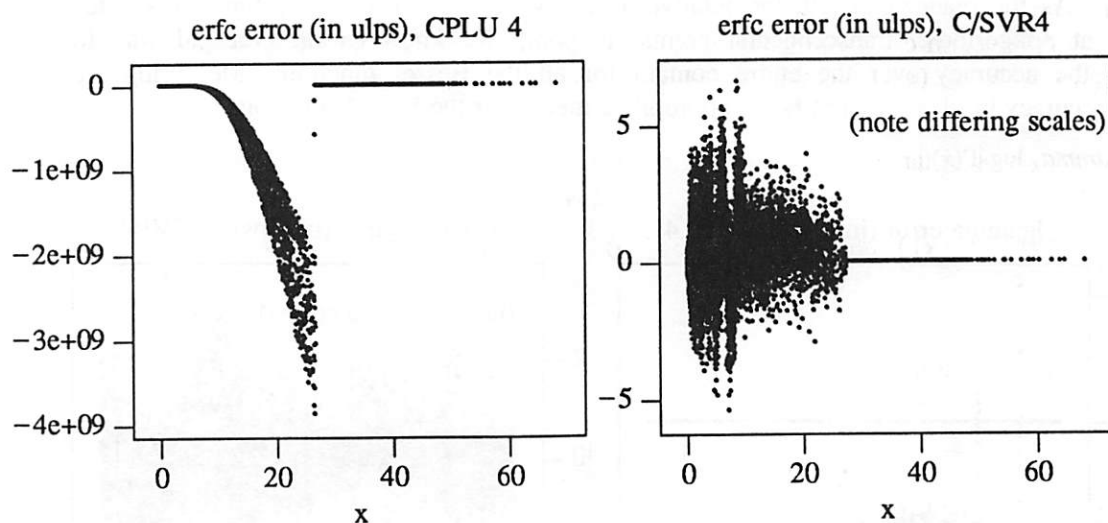
### 6.10 *cosh*

The hyperbolic cosine (*cosh*(*x*)) was improved for C/SVR4 due to a better *exp*(*x*).

### 6.11 *erf*

The error function (*erf*(*x*)) was slightly improved for C/SVR4. See the next section.

### 6.12 *erfc*



The above plots for the complementary error function ( $\text{erfc}(x)$ ) show a large improvement for C/SVR4. One of the standard references for computer approximations ([Hart]) is incomplete and has errors in the error function section. Robert Morris, Sr. developed an improved approximation in the spirit of Hart&Cheney. There is still discernible structure in the error for  $x < 10$  and room for improvement in the approximation.

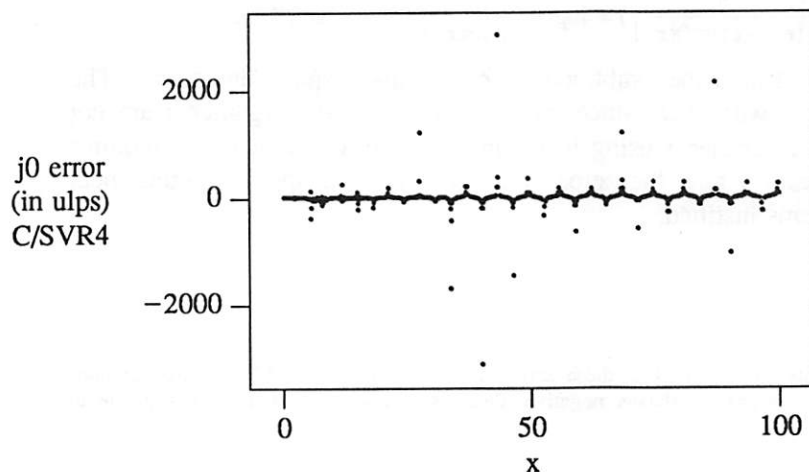
#### 6.13 *exp*

The  $\text{exp}(x)$  function (derived from the 4.3BSD implementation) showed significant improvement.

#### 6.14 *fmod*

The CPLU 4 implementation of the remainder function ( $\text{fmod}(x, y)$ ) was a careful division rather than a modulo operation. The ANSI C standard [ANSI C] requires that  $\text{fmod}(x, y)$  behave as a true modulo. Either the 4.3BSD  $\text{remainder}()$  function or its hardware equivalent (if available) is used to compute the result exactly.

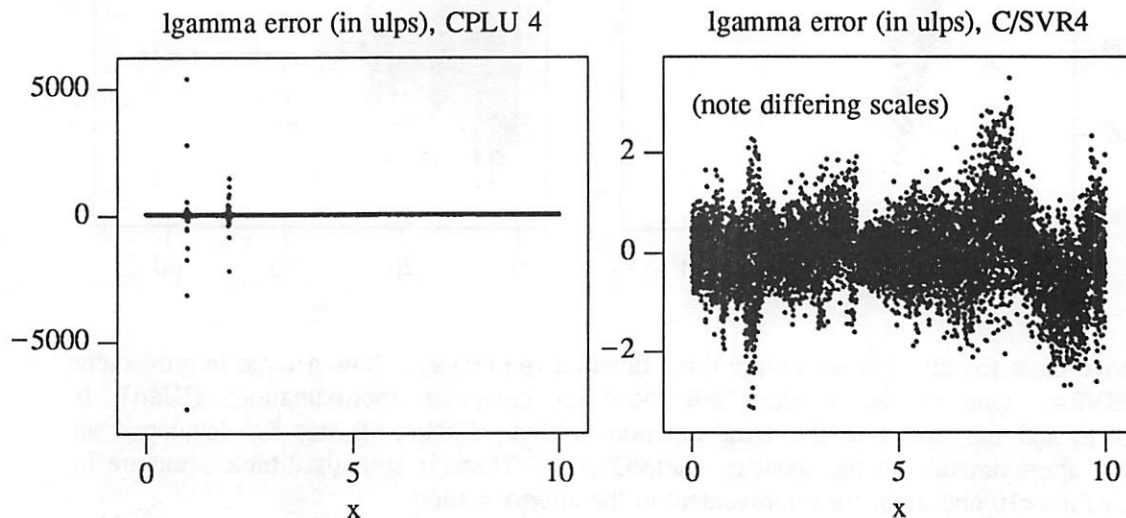
#### 6.15 *j0*



Our only activity for the Bessel functions was to measure the accuracy of  $j_0(x)$  in selected

intervals. As the reader can tell, the relative accuracy of  $j_0(x)$  near the function's zeros (located at nonperiodic, transcendental points) is poor. We know of no practical way to improve the accuracy over the entire domain for all the Bessel functions. Measuring the relative accuracy in ulps may not be an appropriate metric for the Bessel functions.

#### 6.16 *lgamma*, $\log |\Gamma(x)|$



A new implementation of the log of the absolute value of  $\Gamma(x)$  (*lgamma*( $x$ )) is used in C/SVR4. Neither the ulp plots nor the table shows the inaccuracies present in the CPLU 4 implementation for very small arguments. This was caused by a division of a number of order 1 by a denormalized value, causing a floating point overflow. The spikes in the CPLU 4 error plots near  $x=1$ ,  $x=2$  are due to the zeros of the *lgamma*( $x$ ) function at those points. The approximation developed by Cody and Hillstom [Cody-Hill] correctly reproduces the function's logarithmic behavior.

The largest error for C/SVR4 comes for  $x < 0$  near the zeros<sup>12</sup> of *lgamma*( $x$ ). The usual evaluation technique:

$$\text{lgamma}(x) = \log(|\Gamma(x)|) = \log\left(\frac{\pi}{|x\Gamma(-x)\sin(\pi x)|}\right) = \log\left(\frac{\pi}{|x\sin(\pi x)|}\right) - \text{lgamma}(-x)$$

leads to massive cancellation from the subtraction of nearly equal numbers. The Cody&Hillstom technique doesn't work here since the zeros for negative arguments are not conveniently located. To evaluate *lgamma* using techniques we are aware of either requires maintaining over 100 bits of accuracy near the zeros or requires custom approximations near each zero. Neither alternative seems justified.

12. For IEEE double precision arguments, there are 29 of these zeros in the domain  $(-16.1, -2.5)$ . While for non-integer  $x$  smaller than  $-16.1$  *lgamma*( $x$ ) is always negative, there is still some loss of accuracy due to roundoff.

### 6.17 hypot

The accuracy of the euclidean distance function (`hypot(x,y)`) (taken from 4.3BSD) improved significantly.

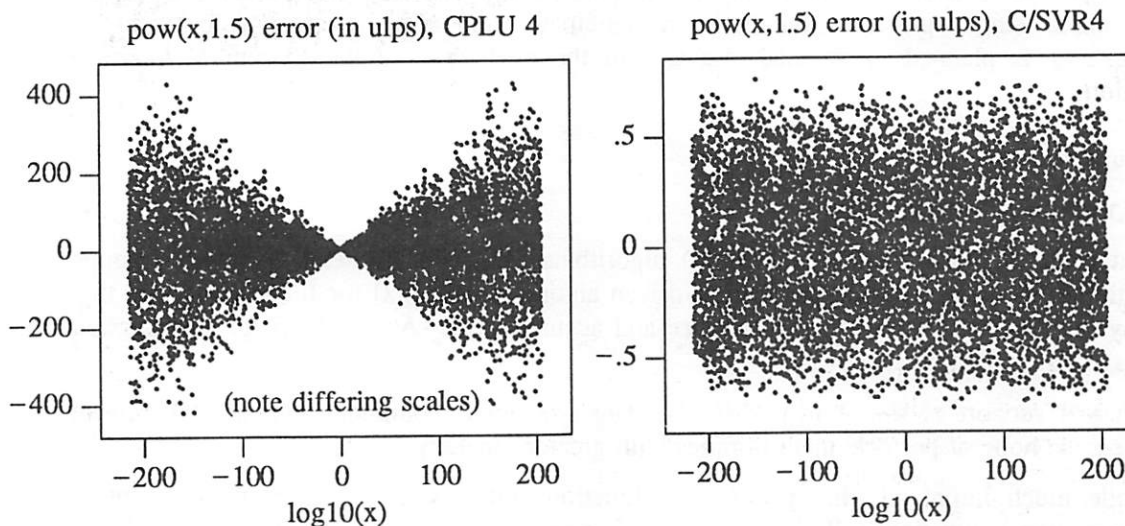
### 6.18 log

The accuracy of the natural logarithm function (`log(x)`) (taken from 4.3BSD) improved significantly.

### 6.19 log10

The base 10 logarithm function (`log10(x)`) showed significant accuracy improvement. `log10(x)` is usually computed from the `log(x)` function using the identity  $\log_{10}(x) = \log(x) / \log(10)$ . Evaluating this product without extra precision results in a larger error for `log10(x)` than for `log(x)`.

### 6.20 pow



The `pow(x,y)` function (taken from 4.3BSD) showed significant improvement. There is still some structure in the ulps plots. The CPLU 4 power function was implemented via

$$\text{pow}(x,y) = x^y = e^{y \log x} = \exp(y * \log(x))$$

Several bits of accuracy are lost by taking the logarithm of the argument, since `log(x)` returns the same result for many different `x`. See Cody&Waite [Cody-Waite] for details.

### 6.21 sin

Changes to the sine function (`sin(x)`) were made as described earlier for the cosine function.

### 6.22 sinh

Some improvement in the hyperbolic sine function (`sinh(x)`) is seen as a result of changes to `exp(x)`.

### 6.23 sqrt

IEEE754 specifies that the square root function (`sqrt(x)`) return the correctly rounded result. The implementation in CPLU 4 incorrectly rounded the results in about 30% of the cases. C/SVR4 corrects this by use of the 4.3BSD `sqrt(x)` function. Accurate hardware-based square root functions exist on several floating point coprocessors and should be used where available.

### 6.24 *tan*

Improvements to the tangent function ( $\tan(x)$ ) were a consequence of changes to the sine and cosine functions.

### 6.25 *tanh*

The changes to the accuracy of the hyperbolic tangent function ( $\tanh(x)$ ) were small.

## 7. Interface changes

The log of the absolute value of  $\Gamma(x)$  was renamed  $\lgamma(x)$  to better describe the function and to match 4.3BSD. Several new functions were added including the inverse trig functions and cube root. A number of single precision functions were added.

Meeting the emerging ANSI C, X/Open and POSIX language and operating system standards required changing some error return values, most notably returning `HUGE_VAL` instead of `HUGE`, where the former is set to infinity (on IEEE754 computers) and the latter was set to the largest finite single precision number. In anticipation of better exception mechanisms, `matherr()` is planned to be made Level 2 in the next issue of the System V Interface Definition.

## 8. Future work

Future work might include:

- Better (or at least a wider choice in) algorithms are needed. Peter Tang at Argonne National Laboratory has designed table-driven algorithms [Tang2] for IEEE computers that may provide both improved performance and accuracy. See [Agar], [Gal] for a discussion of similar techniques.
- Efficient support of the `long double` data type by C compilers and target hardware offers the hope of portable math libraries with greater accuracy.
- While much improved, the `pow(x,y)` function still has larger errors than the other “elementary” functions. Without the use of extended (or effectively extended) precision, we know of no techniques to greatly improve the accuracy.
- Some improvements can still be made to the error functions and the  $\lgamma(x)$  function. These approximations have not been optimized for IEEE754 computers.
- Good accuracy in *directed* rounding modes (now done for `sqrt(x)`) would be useful.
- Standards bodies should reconcile differences in their specification of boundary value behavior (e.g., `pow(0.,0.)`).

We feel that the most needed changes to the math library are to improve the floating point exception mechanism:

- It should be applicable to many computer languages and floating point architectures.
- It should take advantage of IEEE754 facilities if present, but not rely upon them.
- The exception mechanism (including language bindings) should be specified by a standards committee. Lack of standardization has discouraged the use of existing exception mechanisms.
- It should be applicable to developing pipelined and multi-threaded computers.



and to standardize the interfaces to IEEE754 facilities (e.g., directed rounding modes). We look forward to the work of the Numerical C Extensions Group (NCEG) in this area.

## 9. Conclusion

We have measured the accuracy of the UNIX system math library and described the improvements made to many functions for C/SVR4. Any system supporting IEEE754 double precision arithmetic can achieve equal accuracy.

## 10. Acknowledgements

Several people provided assistance during the course of this project. We want to acknowledge the help of Lorinda Cherry, Jim Cody, Barbara Corgan, David Gay, W.Kahan, Andrew Koenig, Alex Liu, Robert Morris, Sr., Matt Nelson, K.C. Ng, Samir Patel, Charles Roberts, Peter Tang, Vic Vyssotsky, and Dave Wolverton.

## References

1. [Agar] R.C. Agarwal, et al., "New Scalar and Vector Elementary Functions for the IBM System/370" IBM J. Res. Develop, **30** March 1986.
2. [ANSI C] ANSI C Draft, December 7, 1988.
3. [Brent] R. P. Brent, "A Fortran Multiple-Precision Arithmetic Package." *ACM Transactions on Mathematical Software*, Vol 4, No. 1, March 1978, pages 57-70.
4. [Cody-Hill] W.J. Cody and K.E. Hillstom, "Chebyshev Approximations for the Natural Logarithm of the Gamma Function," *Mathematics of Computation*, **21**, 1967.
5. [Cody-Waite] William J. Cody, Jr and William Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, 1980.
6. [Gal] S. Gal, "Computing Elementary Functions: A New Approach for Achieving High Accuracy and Good Performance," IBM Technical Report 88.153, March 1985.
7. [Hart] J.F. Hart, et al., *Computer Approximations*. John Wiley & Sons, 1968.
8. [HP] *HP-15C Advanced Functions Handbook*, Hewlett-Packard, 1982.
9. [IEEE754] "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Std. 754-1985, IEEE, 1985.
10. [Payne] Mary Payne and Robert Hanek, "Radian Reduction for Trigonometric Functions," *SIGNUM Newsletter*, **18**, No. 1, January, 1983.
11. [Spaf] Eugene H. Spafford, John C. Flaspohler, "A Report on the Accuracy of Some Floating Point Math Functions on Selected Computers," Georgia Tech. Technical Report GIT-SERC-86/02, GIT-ICS 85/06.
12. [Tang1] Ping Tak Peter Tang, "Some Software Implementations of the Functions Sin and Cos," undated manuscript.
13. [Tang2] Ping Tak Peter Tang, "Portable Implementation of a Generic Exponential Function in Ada," Argonne National Laboratory, ANL-88-3.
14. [Vold] J. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computers*, **EC-8**, 1959. Reprinted in *Computer Arithmetic*, edited by E.E. Swartzlander, Jr., Dowden, Hutchinson & Ross.

## Appendix - Trigonometric Range Reduction

Range reduction for the trig functions is conceptually just a subtraction of a multiple of  $2\pi$  from the argument.

$$X_{reduced} = X - 2\pi n, \quad n \text{ integer}$$

For large values of  $X$ , this subtraction (or the equivalent modulus) may require *hundreds* of significant digits to maintain adequate precision. Several methods of approximate trigonometric range reduction have been used to avoid the computational difficulty of extended precision arithmetic:

**division** Some implementations have simply used double precision arithmetic.

$$X_{reduced} = X - 2\pi_{double}(int)(\frac{X}{2\pi_{double}})$$

$\pi_{double}$  is the best double precision approximation to the transcendental number  $\pi$ .

**extended-division** An improvement to simple division is to use an approximation of  $\pi$  with extra (more than double precision) bits of precision. Cody&Waite [Cody-Waite] describe one such method requiring only double precision operations (used in CPLUS 4). It is also possible to do the division in extended precision floating point where that is available.

Two thresholds **X\_PLOSS** (about  $3.0 \cdot 10^8$ ) and **X\_TLOSS** (about  $1.4 \cdot 10^{16}$ ) were defined to alert users to partial and total loss in accuracy in the range reduction process. Actually, if the argument was close to a multiple of  $\pi/2$ , significant errors in range reduction could occur for much smaller arguments.

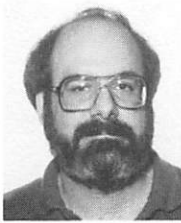
**modulo** An alternative to division is to use a true modulus operation such as the 4.3BSD `remainder(x)` function and  $\pi_{double}$ . (There are some advantages in using an approximation to  $\pi$  to more than double precision accuracy.) This technique is used in some HP calculators [HP] and has the important property of preserving most of the trigonometric identities.

One can argue the relative merits of each approximate range reduction technique. When approximate range reduction is used, the trig function evaluated is effectively a closely related function. This is similar to changing the value of  $\pi$  slightly. Most users would not notice this. We feel that either a modulus (to at least double precision) range reduction or range reduction using true  $\pi$  are appropriate techniques. If the former is chosen, it should be clearly documented to the library user.

We chose to implement true range reduction using the technique described by Payne and Hanek [Payne]. Several thousand bits of  $\pi$  are needed for reducing large arguments. For arguments less than 100, a table-driven range reduction scheme was developed. This provides high accuracy for the trig functions with a loss in performance only for large arguments.

As an exercise in range reduction, one can look at arguments chosen to be very close to multiples of  $\pi/2$ . It can be shown that for IEEE754 double precision arguments greater than one, the argument that returns the smallest result for  $|\sin(x)|$  or  $|\cos(x)|$  is:

$$x = \text{ldexp}(6381956970095103., 797) \quad \text{or about } 5.31937 \cdot 10^{255}$$



**Joel Silverstein**  
AT&T

Joel Silverstein is a Member of the Technical Staff in the Programming Languages Department of AT&T's UNIX(R) Software Operation. He received an M.S. in Computer Science from Columbia University. His current interest involve programming language/operating system interfaces.



**Yio-Chian Tao**  
AT&T

Yio-Chian Tao is a Member of the Technical Staff in the Processor Systems Engineering Department of AT&T Computer Systems. He received an M.S. in Math/Computer Science from Mankato State University, MN. His interest include computer standards, floating point, and computer graphics windowing system.



**Steven Sommars**  
AT&T

Steven Sommars is a Distinguished Member of the Technical Staff in the Processors Systems Engineering Department of AT&T Computer Systems. He received a Ph.D. in Physics from S.U.N.Y Stony Brook. His interests include computer standards, software compatibility and floating point.



# Tcl: An Embeddable Command Language

*John K. Ousterhout*

Computer Science Division  
Electrical Engineering and Computer Sciences  
University of California at Berkeley  
Berkeley, CA 94720  
ouster@sprite.berkeley.edu

## ABSTRACT

Tcl is an interpreter for a tool command language. It consists of a library package that is embedded in tools (such as editors, debuggers, etc.) as the basic command interpreter. Tcl provides (a) a parser for a simple textual command language, (b) a collection of built-in utility commands, and (c) a C interface that tools use to augment the built-in commands with tool-specific commands. Tcl is particularly attractive when integrated with the widget library of a window system: it increases the programmability of the widgets by providing mechanisms for variables, procedures, expressions, etc; it allows users to program both the appearance and the actions of widgets; and it offers a simple but powerful communication mechanism between interactive programs.

## 1. Introduction

Tcl stands for "tool command language". It consists of a library package that programs can use as the basis for their command languages. The development of Tcl was motivated by two observations. The first observation is that a general-purpose programmable command language amplifies the power of a tool by allowing users to write programs in the command language in order to extend the tool's built-in facilities. Among the best-known examples of powerful command languages are those of the UNIX shells [5] and the Emacs editor [8]. In each case a computing environment of unusual power has arisen, in large part because of the availability of a programmable command language.

The second motivating observation is that the number of interactive applications is increasing. In the timesharing environments of the late 1970's and early 1980's almost all programs were batch-oriented. They were typically invoked using an interactive command shell. Besides the shell, only a few other programs needed to be interactive, such as editors and mailers. In contrast, the personal workstations used today, with their raster displays and mice, encourage a different system structure where a large number of programs are interactive and the most common style of interaction is to manipulate individual applications directly with a mouse. Furthermore, the large displays available today make it possible for many interactive applications to be active

---

The work described here was supported in part by the National Science Foundation under Grant ECS-8351961.



at once, whereas this was not practical with the smaller screens of ten years ago.

Unfortunately, few of today's interactive applications have the power of the shell or Emacs command languages. Where good command languages exist, they tend to be tied to specific programs. Each new interactive application requires a new command language to be developed. In most cases application programmers do not have the time or inclination to implement a general-purpose facility (particularly if the application itself is simple), so the resulting command languages tend to have insufficient power and clumsy syntax.

Tcl is an application-independent command language. It exists as a C library package that can be used in many different programs. The Tcl library provides a parser for a simple but fully programmable command language. The library also implements a collection of built-in commands that provide general-purpose programming constructs such as variables, lists, expressions, conditionals, looping, and procedures. Individual application programs extend the basic Tcl language with application-specific commands. The Tcl library also provides a set of utility routines to simplify the implementation of tool-specific commands.

I believe that Tcl is particularly useful in a windowing environment, and that it provides two advantages. First, it can be used as a general-purpose mechanism for programming the interfaces of applications. If a tool is based on Tcl, then it should be relatively easy to modify the application's user interface and to extend the interface with new commands. Second, and more important, Tcl provides a uniform framework for communication between tools. If used uniformly in all tools, Tcl will make it possible for tools to work together more gracefully than is possible today.

The rest of this paper is organized as follows. Section 2 describes the Tcl language as seen by users. Section 3 discusses how Tcl is used in applications, including the C-language interface between application programs and the Tcl library. Section 4 describes how Tcl can be used in a windowing environment to customize interface actions and appearances. Section 5 shows how Tcl can be used as a vehicle for communication between applications, and why this is important. Section 6 presents the status of the Tcl implementation and some preliminary performance measurements. Section 7 compares Tcl to Lisp, Emacs, and NeWS, and Section 8 concludes the paper.

## 2. The Tcl Language

In a sense, the syntax of the Tcl language is unimportant: any programming language, whether it is C [6], Forth [4], Lisp [1], or Postscript [2], could provide many of the same programmability and communication advantages as Tcl. This suggests that the best implementation approach is to borrow an existing language and concentrate on providing a convenient framework for the use of that language. However, the environment for an embeddable command language presents an unusual set of constraints on the language, which are described below. I eventually decided that a new language designed from scratch could probably meet the constraints with less implementation effort than any existing language.

Tcl is unusual because it presents two different interfaces: a textual interface to users who issue Tcl commands, and a procedural interface to the applications in which it is embedded. Each of these interfaces must be simple, powerful, and efficient. There were four major factors in the language design:

- [1] **The language is for commands.** Almost all Tcl "programs" will be short, many only one line long. Most programs will be typed in, executed once or perhaps a few times, and then discarded. This suggests that the language should have a simple syntax so that it is easy to type commands. Most existing programming languages have complex syntax; the syntax is helpful when writing long programs but would be clumsy if used for a command

language.

- [2] **The language must be programmable.** It should contain general programming constructs such as variables, procedures, conditionals, and loops, so that users can extend the built-in command set by writing Tcl procedures. Extensibility also argues for a simple syntax: this makes it easier for Tcl programs to generate other Tcl programs.
- [3] **The language must permit a simple and efficient interpreter.** For the Tcl library to be included in many small programs, particularly on machines without shared-library facilities, the interpreter must not occupy much memory. The mechanism for interpreting Tcl commands must be fast enough to be usable for events that occur hundreds of times a second, such as mouse motion.
- [4] **The language must permit a simple interface to C applications.** It must be easy for C applications to invoke the interpreter and easy for them to extend the built-in commands with application-specific commands. This factor was one of the reasons why I decided not to use Lisp as the command language: Lisp's basic data types and storage management mechanisms are so different than those of C that it would be difficult to build a clean and simple interface between them. For Tcl I used a data type (string) that is natural to C.

## 2.1. Tcl Language Syntax

Tcl's basic syntax is similar to that of the UNIX shells: a command consists of one or more fields separated spaces or tabs. The first field is the name of a command, which may be either a built-in command, an application-specific command, or a procedure consisting of a sequence of Tcl commands. Fields after the first one are passed to the command as arguments. Newline characters are used as command separators, just as in the UNIX shells, and semi-colons may be used to separate commands on the same line. Unlike the UNIX shells, each Tcl command returns a string result, or the empty string if a return value isn't appropriate.

There are four additional syntactic constructs in Tcl, which give the language a Lisp-like flavor. Curly braces are used to group complex arguments; they act as nestable quote characters. If the first character of an argument is a open brace, then the argument is not terminated by white space. Instead, it is terminated by the matching close brace. The argument passed to the command consists of everything between the braces, with the enclosing braces stripped off. For example, the command

```
set a {dog cat {horse cow mule} bear}
```

will receive two arguments: "a" and "dog cat {horse cow mule} bear". This particular command will set the variable `a` to a string equal to the second argument. If an argument is enclosed in braces, then none of the other substitutions described below is made on the argument. One of the most common uses of braces is to specify a Tcl subprogram as an argument to a Tcl command.

The second syntactic construct in Tcl is square brackets, which are used to invoke command substitution. If an open bracket appears in an argument, then everything from the open bracket up to the matching close bracket is treated as a command and executed recursively by the Tcl interpreter. The result of the command is then substituted into the argument in place of the bracketed string. For example, consider the command

```
set a [format {Santa Claus is %s years old} 99]
```

The `format` command does `printf`-like formatting and returns the string "Santa Claus is 99 years old", which is then passed to `set` and assigned to variable `a`.

The third syntactic construct is the dollar sign, which is used for variable substitution. If it appears in an argument then the following characters are treated as a variable name; the contents of the variable are substituted into the argument in place of the dollar sign and name. For

example, the commands

```
set b 99
set a [format {Santa Claus is %s years old} $b]
```

result in the same final value for `a` as the single command in the previous paragraph. Variable substitution isn't strictly necessary since there are other ways to achieve the same effect, but it reduces typing.

The last syntactic construct is the backslash character, which may be used to insert special characters into arguments, such as curly braces or non-printing characters.

## 2.2. Data Types

There is only one type of data in Tcl: strings. All commands, arguments to commands, results returned by commands, and variable values are ASCII strings. The use of strings throughout Tcl makes it easy to pass information back and forth between Tcl library procedures and C code in the enclosing application. It also makes it easier to pass Tcl-related information back and forth between machines of different types.

Although everything in Tcl is a string, many commands expect their string arguments to have particular formats. There are three particularly common formats for strings: lists, expressions, and commands. A list is just a string containing one or more fields separated by white space, similar to a command. Curly braces may be used to enclose complex list elements; these complex list elements are often lists in their own right, as in Lisp. For example, the string

```
dog cat {horse cow mule} bear
```

is a list with four elements, the third of which is a list with three elements. Tcl provides commands for a number of list-manipulation operations, such as creating lists, extracting elements, and computing list lengths.

The second common form for a string is a numeric expression. Tcl expressions have the same operators and precedence as expressions in C. The `expr` Tcl command evaluates a string as an expression and returns the result (as a string, of course). For example, the command

```
expr {($a < $b) || ($c != 0)}
```

returns "1" if the numeric value of variable `a` is less than that of variable `b`, or if variable `c` is zero; otherwise it returns "0". Several other commands, such as `if` and `for`, expect one or more of their arguments to be expressions.

The third common interpretation of strings is as commands (or sequences of commands). Arguments of this form are used in Tcl commands that implement control structures. For example, consider the following command:

```
if {$a < $b} {
    set tmp $a
    set a $b
    set b $tmp
}
```

The `if` command receives two arguments here, each of which is delimited by curly braces. If is a built-in command that evaluates its first argument as an expression; if the result is non-zero, `if` executes its second argument as a Tcl command. This particular command swaps the values of the variables `a` and `b` if `a` is less than `b`.

Tcl also allows users to define command procedures written in the Tcl language. I will refer to these procedures as *tclproc*'s, in order to distinguish them from other procedures written in C. The `proc` built-in command is used to create a *tclproc*. For example, here is a Tcl command

that defines a recursive factorial procedure:

```
proc fac x {
    if {$x == 1} {return 1}
    return [expr {$x * [fac [expr $x-1]]}]
}
```

The `proc` command takes three arguments: a name for the new tclproc, a list of variable names (in this case the list has only a single element, `x`), and a Tcl command that comprises the body of the tclproc. Once this `proc` command has been executed, `fac` may be invoked just like any other Tcl command. For example

```
fac 4
```

will return the string "24".

Figure 1 lists all of the built-in Tcl commands in groups. In addition to the commands already mentioned, Tcl provides commands for manipulating strings (comparison, matching, and `printf/scanf`-like operations), commands for manipulating files and file names, and a command to fork a subprocess and return the subprocess's standard output as result. The built-in Tcl commands provide a simple but complete programming language. The built-in facilities may be

<b>Control</b>
break, case, continue, eval, for, foreach, if
<b>Variables and Procedures</b>
global, proc, return, set
<b>List Manipulation</b>
concat, index, length, list, range
<b>Expressions</b>
expr
<b>String Manipulation</b>
format, scan, string
<b>File Manipulation</b>
file, glob, print, source
<b>Invoking Subprocesses</b>
exec
<b>Miscellaneous</b>
catch, error, info, time

**Figure 1.** The built-in Tcl commands. This set of commands is available to any application that uses Tcl. Additional commands may be defined by the application.



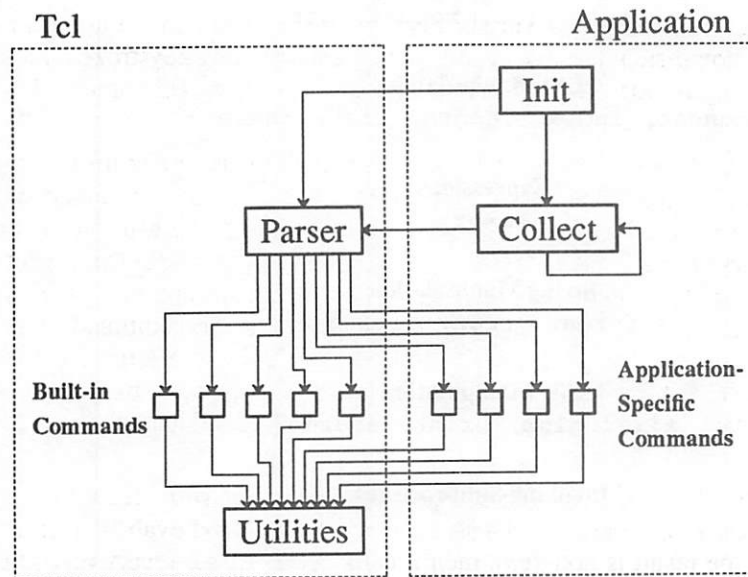
extended in three ways: by writing tclprocs; by invoking other programs as subprocesses; or by defining new commands with C procedures as described in the next section.

### 3. Embedding Tcl in Applications

Although the built-in Tcl commands could conceivably be used as a stand-alone programming system, Tcl is really intended to be embedded in application programs. I have built several application programs using Tcl, one of which is a mouse-based editor for X called *mx*. In the rest of the paper I will use examples from *mx* to illustrate how Tcl interacts with its enclosing application.

An application using Tcl extends the built-in commands with a few additional commands related to that particular application. For example, a clock program might provide additional commands to control how the clock is displayed and to set alarms; the *mx* editor provides additional commands to read a file from disk, display it in a window, select and modify ranges of bytes, and write the modified file back to disk. An application programmer need only write the application-specific commands; the built-in commands provide programmability and extensibility "for free". To users, the application-specific commands appear the same as the built-in commands.

Figure 2 shows the relationship between Tcl and the rest of an application. Tcl is a C library package that is linked with the application. The Tcl library includes a parser for the Tcl language, procedures to execute the built-in commands, and a set of utility procedures for things like expression evaluation and list management. The parser includes an extension interface that



**Figure 2.** The Tcl library provides a parser for the Tcl language, a set of built-in commands, and several utility procedures. The application provides application-specific commands plus procedures to collect commands for execution. The commands are parsed by Tcl and then passed to relevant command procedures (either in Tcl or in the application) for execution.



may be used to extend the language's command set.

To use Tcl, an application first creates an object called an *interpreter*, using the following library procedure:

```
Tcl_Interp * Tcl_CreateInterp()
```

An interpreter consists of a set of commands, a set of variable bindings, and a command execution state. It is the basic unit manipulated by most of the Tcl library procedures. Simple applications will use only a single interpreter, while more complex applications may use multiple interpreters for different purposes. For example, *mx* uses one interpreter for each window on the screen.

Once an application has created an interpreter, it calls the `Tcl_CreateCommand` procedure to extend the interpreter with application-specific commands:

```
typedef int (*Tcl_CmdProc)(ClientData clientData,
    Tcl_Interp *interp, int argc, char *argv[]);

Tcl_CreateCommand(Tcl_Interp *interp, char *name,
    Tcl_CmdProc proc, ClientData clientData)
```

Each call to `Tcl_CreateCommand` associates a particular command name (*name*) with a procedure that implements that command (*proc*) and an arbitrary single-word value to pass to that procedure (*clientData*).

After creating application-specific commands, the application enters a main loop that collects commands and passes them to the `Tcl_Eval` procedure for execution:

```
int Tcl_Eval(Tcl_Interp *interp, char *cmd)
```

In the simplest form, an application might simply read commands from the terminal or from a file. In the *mx* editor Tcl commands are associated with events such as keystrokes, mouse buttons, or menu activations; each time an event occurs, the corresponding Tcl command is passed to `Tcl_Eval`.

The `Tcl_Eval` procedure parses its *cmd* argument into fields, looks up the command name in the table of those associated with the interpreter, and invokes the command procedure associated with that command. All command procedures, whether built-in or application-specific, are called in the same way, as described in the typedef for `Tcl_CmdProc` above. A command procedure is passed an array of strings describing the command's arguments (*argc* and *argv*) plus the *clientData* value that was associated with the command when it was created. *ClientData* is typically a pointer to an application-specific structure containing information needed to execute the command. For example, in *mx* the *clientData* argument points to a per-window data structure describing the file being edited and the window it is displayed in.

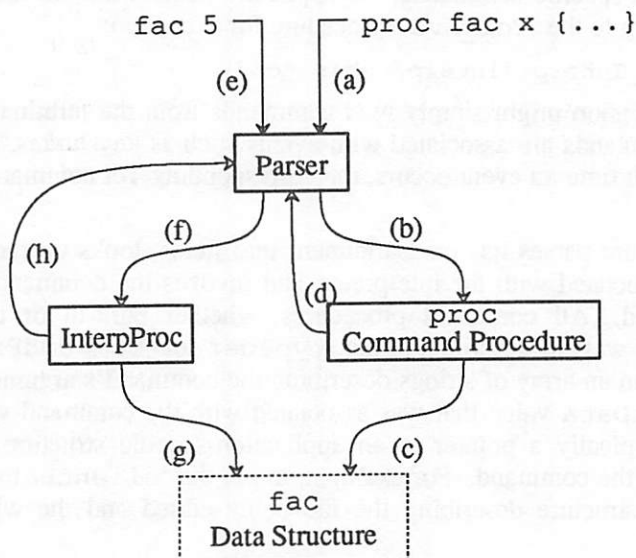
Control mechanisms like *if* and *for* are implemented with recursive calls to `Tcl_Eval`. For example, the command procedure for the *if* command evaluates its first argument as an expression; if the result is non-zero, then it calls `Tcl_Eval` recursively to execute its second argument as a Tcl command. During the execution of that command, `Tcl_Eval` may be called recursively again, and so on. `Tcl_Eval` also calls itself recursively to execute bracketed commands that appear in arguments.

Even tclprocs such as *fac* use this same basic mechanism. When the *proc* command is invoked to create *fac*, the *proc* command procedure creates a new command by calling `Tcl_CreateCommand` as illustrated in Figure 3. The new command has the name *fac*. Its command procedure (*proc* in the call to `Tcl_CreateCommand`) is a special Tcl library procedure called `InterpProc`, and its *clientData* is a pointer to a structure describing the

`tclproc`. This structure contains, among other things, a copy of the body of the `tclproc` (the third argument to the `proc` command). When the `fac` command is invoked, `Tcl_Eval` calls `InterpProc`, which in turn calls `Tcl_Eval` to execute the body of the `tclproc`. There is some additional code required to associate the argument of the `fac` command (which is passed to `InterpProc` in its `argv` array) with the `x` variable used inside `fac`'s body, and to support variables with local scope, but much of the mechanism for `tclprocs` is the same as that for any other Tcl command.

A Tcl command procedure returns two results to `Tcl_Eval`: an integer return code and a string. The return code is returned as the procedure's result, and the string is stored in the interpreter, from which it can be retrieved later. `Tcl_Eval` returns the same code and string to its caller. Table I summarizes the return codes and strings. Normally the return code is `TCL_OK` and the string contains the result of the command. If an error occurs in executing a command, then the return code will be `TCL_ERROR` and the string will describe the error condition. When `TCL_ERROR` is returned (or any value other than `TCL_OK`), the normal action is for nested command procedures to return the same code and string to their callers, unwinding all pending command executions until eventually the return code and string are returned by the top-level call to `Tcl_Eval`. At this point the application will normally display the error message for the user by printing it on the terminal or displaying it in a notifier window.

Return codes other than `TCL_OK` or `TCL_ERROR` cause partial unwinding. For example, the `break` command returns a `TCL_BREAK` code. This causes nested command executions to be unwound until a nested `for` or `foreach` command is reached. When a `for` or



**Figure 3.** The creation and execution of a `tclproc` (a procedure written in Tcl): (a) the `proc` command is invoked, e.g. to create the `fac` procedure; (b) the Tcl parser invokes the command procedure associated with `proc`; (c) the `proc` command procedure creates a data structure to hold the Tcl command that is `fac`'s body; (d) `fac` is registered as a new Tcl command, with `InterpProc` as its command procedure; (e) `fac` is invoked as a Tcl command; (f) the Tcl parser invokes `InterpProc` as the command procedure for `fac`; (g) `InterpProc` retrieves the body of `fac` from the data structure; and (h) the Tcl commands in `fac`'s body are passed back to the Tcl parser for execution.

Return Code	Meaning	String
TCL_OK	Command completed normally	Result
TCL_ERROR	Error occurred in command	Error message
TCL_BREAK	Should abort innermost loop	None
TCL_CONTINUE	Should skip innermost iteration	None
TCL_RETURN	Should return from innermost procedure	Procedure result

**Table I.** Each Tcl command returns a code describing what happened and a string that provides additional information. If the return code is not `TCL_OK`, then nested command executions unwind and return the same code, until reaching top-level or some command that is prepared to deal with the exceptional return code.

`foreach` command invokes `Tcl_Eval` recursively, it checks specially for the `TCL_BREAK` result. When this occurs the `for` or `foreach` command terminates the loop, but it doesn't return the `TCL_BREAK` code to its caller. Instead it returns `TCL_OK`. Thus no higher levels of execution are aborted. The `TCL_CONTINUE` return code is also handled by the `for` and `foreach` commands (they go on to the next loop iteration) and `TCL_RETURN` is handled by the `InterpProc` procedure. Only a few command procedures, like `break` and `for`, know anything about special return codes such as `TCL_BREAK`; other command procedures simply abort whenever they see any return code other than `TCL_OK`.

The `catch` command may be used to prevent complete unwinding on `TCL_ERROR` returns. `Catch` takes an argument that is a Tcl command to execute. It passes the command to `Tcl_Eval` for execution, but always returns `TCL_OK`. If an error occurs in the command, `catch`'s command procedure detects the `TCL_ERROR` return value from `Tcl_Eval`, saves information about the error in Tcl variables, and then returns `TCL_OK` to its caller. In almost all cases I think the best response to an error is to abort all command invocations and notify the user; `catch` is provided for those few occasions where an error is expected and can be handled without aborting.

#### 4. Tcl and Window Applications

An embeddable command language like Tcl offers particular advantages in a windowing environment. This is partly because there are many interactive programs in a windowing environment (hence many places to use a command language) and partly because configurability is important in today's windowing environments and a language like Tcl provides the flexibility to reconfigure. Tcl can be used for two purposes in a window application: to configure the application's interface *actions*, and to configure the application's interface *appearance*. These two purposes are discussed in the paragraphs below.

The first use of Tcl is for interface actions. Ideally, each event that has any importance to the application should be bound to a Tcl command. Each keystroke, each mouse motion or mouse button press (or release), and each menu entry should be associated with a Tcl command. When the event occurs, it is first mapped to its Tcl command and then executed by passing the command to `Tcl_Eval`. The application should not take any actions directly; all actions should first pass through Tcl. Furthermore, the application should provide Tcl commands that allow the user to change the Tcl command associated with any event.

In interactive windowing applications, the use of Tcl will probably not be visible to beginning users: they will manipulate the applications using buttons, menus, and other interface components. However, if Tcl is used as an intermediary for all interface actions then two advantages accrue. First, it becomes possible to write Tcl programs to reconfigure the interface. For



example, users will be able to rebind keystrokes, change mouse buttons, or replace an existing operation with a more complex one specified as a set of Tcl commands or `tclprocs`. The second advantage is that this approach forces all of the application's functionality to be accessible through Tcl: anything that can be invoked with the mouse or keyboard can also be invoked with Tcl programs. This makes it possible to write `tclprocs` that simulate the actions of the program, or that compose the program's basic actions into more powerful actions. It also permits interactive sessions to be recorded and replayed as a sequence of Tcl commands (see Section 5).

The second use for Tcl in a window application is to configure the appearance of the application. All of the application's interface components ("widgets" in X terminology), such as labels, buttons, text entries, menus, and scrollbars, should be configured using Tcl commands. For example, in the case of a button the application (or the button widget code) should provide Tcl commands to change the button's size and location, its text, its colors, and the action (a Tcl command, of course) to invoke when the button is activated. This makes it possible for users to write Tcl programs to personalize the layout and appearance of the applications they use. The most common use of such reconfigurability would probably be in Tcl command files read by programs automatically when they start execution. However, the Tcl commands could also be used to change an application's appearance while it is running, if that should prove useful.

If Tcl is used as described above, then it could serve as a specification language for user interfaces. User interface editors could be written to display widgets and let users re-arrange them and configure attributes such as colors and associated Tcl commands. The interface editor could then output information about the interface as a Tcl command file to be read by the application when it starts up. Some current interface editors output C code which must then be compiled into the application [7]; unfortunately this approach requires an application to be recompiled in order to change its interface (or, alternatively, it requires a dynamic-code-loading facility). If Tcl were used as the interface specification language then no recompilation would be necessary and a single application binary could support many different interfaces.

## 5. Communication Between Applications

The advantages of an embedded command language like Tcl become even greater if all of the tools in an environment are based on the same language. First, users need only learn one basic command language; to move from one application to another they need only learn the (few?) application-specific commands for the new application. Second, generic interface editors become possible, as described in the previous section. Third, and most important in my view, Tcl can provide a means of communication between applications.

I have implemented a communication mechanism for X11 in the form of an additional Tcl command called `send`. For `send` to work, each Tcl interpreter associated with an X11 application is given a textual name, such as `xmh` for an X mail handler or `mx.foo.c` for a window in which `mx` is displaying a file named `foo.c`. The `send` command takes two arguments: the name of an interpreter and a Tcl command to execute in that interpreter. `Send` arranges for the command to be passed to the process containing the named interpreter; the command is executed by that interpreter and the results (return code and string) are returned to the application that issued the `send` command.

The X11 implementation of `send` uses a special property attached to the root window. The property stores the names of all the interpreters plus a window identifier for each interpreter. A command is sent to an interpreter by appending it to a particular property in the interpreter's associated window. The property change is detected by the process that owns the interpreter; it reads the property, executes the command, and appends result information onto a property associated with the sending application. Finally, the sending application detects this change of property, reads the result information, and returns it as the result of the `send` command.

The `send` command provides a powerful way for one application to control another. For example, a debugger could send commands to an editor to highlight the current source line as it single-steps through a program. Or, a user interface editor could use `send` to manipulate an application's interface directly: rather than modifying a dummy version of the application's interface displayed by the interface editor, the interface editor could use `send` to modify the interface of a "live" application, while also saving the configuration for a configuration file. This would allow an interface designer to try out the look and feel of a new interface incrementally as changes are made to the interface.

Another example of using `send` is for changing user preferences. If one user walks up to a display that has been configured for some other user, the new user could run a program that finds out about all the existing applications on the screen (by querying the property that contains their names), reads the new user's configuration file for each application, and sends commands to that application to reconfigure it for the new user's preferences. When the old user returns, he or she could invoke the same program to restore the original preferences.

`Send` could also be used to record interactive sessions involving multiple applications and then replay the sessions later (e.g. for demonstration purposes). This would require an additional Tcl command called `trace`; `trace` would take a single argument (a Tcl command string) and cause that command string to be executed before each other command was executed in that interpreter. Within a single application, `trace` could be used to record each Tcl command before it is executed, so that the commands could be replayed later. In a multi-application environment, a recorder program could be built using `send`. The recorder sends a `trace` command to each application to be recorded. The `trace` command arranges for information to be sent back to the recorder about each command executed in that application. The recorder then logs information about which applications executed which commands. The recorder can re-execute the commands by `send`-ing them back to the applications again. The `trace` command does not yet exist in Tcl, but it could easily be added.

`Send` provides a much more powerful mechanism for communication between applications than is available today. The only easy-to-use form of communication for today's applications is the selection or cut buffer: a single string of text that may be set by one application and read by another. `Send` provides a more general form of communication akin to remote procedure call [3]. If all of an application's functionality is made available through Tcl, as described in Section 4, then `send` makes all of each application's functionality available to other applications as well.

If Tcl (and `send`) were to become widely used in window applications, I believe that a better kind of interactive environment would arise, consisting of a large number of small specialized applications rather than a few monolithic ones. Today's applications cannot communicate with each other very well, so each application must incorporate all the functionality that it needs. For example, some window-based debuggers contain built-in text editors so that they can highlight the current point of execution. With Tcl and `send`, the debugger and the editor could be distinct programs, with each `send`-ing commands to the other as necessary. Ideally, monolithic applications could be replaced by lots of small applications that work together in exciting new ways, just as the UNIX shells allowed lots of small text processing applications to be combined together. I think that Tcl, or some other language like it, will provide the glue that binds together the windowing applications of the 1990's.

## 6. Status and Performance

The Tcl language was designed in the fall of 1987 and implemented in the winter of 1988. In the spring of 1988 I incorporated Tcl into the *mx* editor (which already existed, but with an inferior command language), and also into a companion terminal emulator called *Tx*. Both of



these programs have been in use by a small user community at Berkeley for the last year and a half. All of the Tcl language facilities exist as described above, except that the `send` command is still in prototype form and `trace` hasn't been implemented. Some of the features described in Section 4, such as menu and keystroke bindings, are implemented in *mx*, but in an *ad hoc* fashion: Tcl is not yet integrated with a widget set. I am currently building a new toolkit and widget set that is based entirely on Tcl. When it is completed, I expect it to provide all of the features described in Section 4. As of this writing, the implementation has barely begun.

Table II shows how long it takes Tcl to execute various commands on two different workstations. On Sun-3 workstations, the average time for simple commands is about 500 microseconds, while on DECstation 3100's the average time per command is about 160 microseconds. Although *mx* does not currently use a Tcl command for each mouse motion event, the times in Table II suggest that this would be possible, even on Sun-3 workstations, without significant degradation of response. For example, if mouse motion events occur 100 times per second, the Tcl overhead for dispatching one command per event will consume only about 1-2% of a Sun-3 processor. For the ways in which Tcl is currently used (keystroke and menu bindings consisting of a few commands), there are no noticeable delays associated with Tcl. For application-specific commands such as those for the *mx* editor, the time to execute the command is much greater than the time required by Tcl to parse it and call the command procedure.

The Tcl library is small enough to be used in a wide variety of programs, even on systems without mechanisms for sharing libraries. The Tcl code consists of about 7000 lines of C code (about half of which is comments). When compiled for a Motorola 68000, it generates about 27000 bytes of object code.

## 7. Comparisons

The Tcl language has quite a bit of surface similarity to Lisp, except that Tcl uses curly braces or brackets instead of parentheses and no braces are needed around the outermost level of a command. The greatest difference between Tcl and Lisp is that Lisp evaluates arguments by default, whereas in Tcl arguments are not evaluated unless surrounded by brackets. This means that more typing effort is required in Tcl if an argument is to be evaluated, and more typing effort is required in Lisp if an argument is to be quoted (not evaluated). It appeared to me that no-

Tcl Command	Sun-3 Time (microseconds)	DS3100 Time (microseconds)
<code>set a 1</code>	225	57
<code>list abc def ghi jkl</code>	460	138
<code>if {4 &gt; 3} {set a 1}</code>	700	220
<pre>proc fac x {     if {\$x == 1} {return 1}     return [expr {\$x*[fac [expr \$x-1]]}] }</pre>	1280	380
<code>fac 5</code>	11250	3630

**Table II.** The cost of various Tcl commands, measured on a Sun-3/75 workstation and on a DECstation 3100. The command `fac 5` executes a total of 23 Tcl commands, for an average command time of about 500 microseconds on a Sun-3 or 160 microseconds on a DECstation 3100.

evaluation is usually the desired result in arguments to a command language, so I made this the default in Tcl. Tcl also has fewer data types than Lisp; this was done in order to simplify the interface between the Tcl library and an enclosing C application.

The Emacs editor is similar to Tcl in that it provides a framework that can be used to control many different application programs. For example, subprocesses can be run in Emacs windows and users can write Emacs command scripts that (a) generate command sequences for input to the applications and (b) re-format the output of applications. This allows users to embellish the basic facilities of applications, edit their output, and so on. The difference between Emacs and Tcl is that the programmability is centralized in Emacs: applications cannot talk to each other unless Emacs acts as intermediary (e.g. to set up a new communication mechanism between two applications, code must be written in Emacs to pass information back and forth between the applications). The Tcl approach is decentralized: each application has its own command interpreter and applications may communicate directly with each other.

Lastly, it is interesting to compare Tcl to NeWS [9], a window system that is based on the Postscript language. NeWS allows applications to download Postscript programs into the window server in order to change the user interface and modify other aspects of the system. In a sense, this is similar to the `send` command in Tcl, in that applications may send programs to the server for execution. However, the NeWS mechanism is less general than Tcl: NeWS applications generate Postscript programs as output but they do not necessarily respond to Postscript programs as input. In other words, NeWS applications can affect each others' interfaces, by controlling the server, but they cannot directly invoke each others' application-specific operations as they can with Tcl.

To summarize, the Tcl approach is less centralized than either the Emacs or NeWS approaches. For a windowing environment with large numbers of independent tools, I think the decentralized approach makes sense. In fairness to Emacs, it's important to point out that Emacs wasn't designed for this environment, and that Emacs works quite nicely in the environment for which it was designed (ASCII terminals with batch-style applications). It's also worth noting that direct communication between applications was not an explicit goal of the NeWS system design.

## 8. Conclusions

I think that Tcl could improve our interactive environments in three general ways. First, Tcl can be used to improve individual tools by providing them with a programmable command language; this allows users to customize tools and extend their functionality. Second, Tcl can provide a uniform command language across a range of tools; this makes it easier for users to program the tools and also allows tool-independent facilities to be built, such as interface editors. Third, Tcl provides a mechanism for tools to control each other; this encourages a more modular approach to windowing applications and makes it possible to re-use old applications in new ways. In my opinion the third benefit is potentially the most important.

My experiences with Tcl so far are positive but limited. Tcl needs a larger user community and a more complete integration into a windowing toolkit before it can be fully evaluated. The Tcl library source code is currently available to the public in a free, unlicensed form, and I hope to produce a Tcl-based toolkit in the near future.

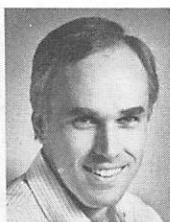
## 9. Acknowledgments

The members of the Sprite project acted as guinea pigs for the editor and terminal emulator based on Tcl; without their help the language would not have evolved to its current state. Fred Douglass, John Hartman, Ken Shirriff, and Brent Welch provided helpful comments that improved

the presentation of this paper.

## 10. References

- [1] Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.
- [2] Adobe Systems, Inc. *Postscript Language Tutorial and Cookbook*, Addison-Wesley, Reading, MA, 1985.
- [3] Birrell, A. and Nelson, B. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1986, pp. 39-59.
- [4] Brodie, L. *Starting FORTH: An Introduction to the FORTH Language and Operating System for Beginners and Professionals*, Prentice Hall, Englewood Cliffs, NJ, 1981.
- [5] Kernighan, B.W. and Pike, R. *The UNIX Programming Environment*, Prentice Hall, Englewood Cliffs, NJ, 1984.
- [6] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*, Second Edition, Prentice Hall, Englewood Cliffs, NJ, 1988.
- [7] Mackey, K., Downs, M., Duffy, J., and Leege, J. "An Interactive Interface Builder for Use with Ada Programs," *Xhibition Conference Proceedings*, 1989.
- [8] Stallman, R. *GNU Emacs Manual*, Fourth Edition, Version 17, February 1986.
- [9] Sun Microsystems, Inc. *NeWS Technical Overview*, Sun Microsystems, Inc. PN 800-1498-05, 1987.



**John K. Ousterhout**  
UC Berkeley

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He and his students have developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is now leading the development of Sprite, a network operating system for high-performance workstations. Ousterhout is a recipient of the ACM Grace Murray Hopper Award,

the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.

# An Event-based Fair Share Scheduler

Raymond B. Essick  
Prisma, Inc.  
essick@prisma.com

## Abstract

The PrismaOS scheduler is an event-driven scheduler with fair share features. We implemented this scheduler to replace the standard SunOS scheduler for the Prisma P1 system, which is expected to accommodate thousands of processes. Our goals and reasons for implementing a new scheduler in PrismaOS instead of using the standard SunOS scheduler included: improved interactive responsiveness, scaling to support thousands of processes efficiently, and fair share scheduling.

This paper describes the PrismaOS scheduler, a modification of the ESCHED scheduler. It briefly covers the changes made to the basic ESCHED algorithms and then details the implementation of a fair share scheduler on the ESCHED base. The fair share discussion describes our early implementation and the tuning steps we went through to produce our final fair share scheduler for PrismaOS.

The new scheduler, in its various forms, has been running in production on approximately one dozen machines at Prisma since November 1988.

## 1. Introduction

For the Prisma P1, we wanted a scheduler that would behave well in a system with many processes. The standard SunOS scheduler scans all processes every second to recalculate priorities. In a system with thousands of processes, the overhead of this periodic recalculation is unacceptable. We determined that using *events* was a better base for making scheduling decisions than periodic priority recalculations.

We also saw a need for a Fair Share scheduler in our system. A Fair Share scheduler allows a system administrator to apportion the machine among user groups so that the groups are guaranteed to receive specified percentages of the machine. We examined several existing UNIX fair share schedulers, but all of them shared the same underlying scheduler as standard SunOS and therefore did not meet our goals.

Our overall goals in developing the scheduler can be summarized as follows:

- We wanted to improve system responsiveness for interactive users.
- We wanted algorithms that would scale well to environments with thousands of processes.
- We wanted a Fair Share scheduler.
- We wanted to make efficient use of the machine (i.e., low overhead).

The first part of our approach was to implement an event-based scheduler in our kernel; we chose to use the ESCHED scheduler [Straathof86] as a starting point because it has a previous record as a workable event scheduler in a UNIX kernel. We then proceeded to address what we saw as algorithmic shortcomings in the ESCHED scheduler. Once we had our event scheduler, we looked at the algorithms needed to implement fair share scheduling using that scheduler.

In this paper, we describe our algorithmic changes to the ESCHED scheduler and the algorithms we used to implement a fair share scheduler. We also discuss how we modified our initial fair share algorithms as we gained experience running the scheduler on our production systems.

The paper is divided into several sections. Section 2 discusses the event-based scheduler and our enhancements. Section 3 discusses fair share schedulers and our initial fair share algorithms. Section 4 describes the tools we developed to interact with and control the scheduler. Section 5 describes our observations and improvements to the scheduling algorithms. Section 6 discusses plans for additional work on the scheduler. Section 7 summarizes our findings.



## 2. Event Scheduling

Event scheduling is an approach where process priorities are recalculated only when specific *events* occur. This differs from the standard UNIX scheduler, which recalculates priorities for all processes every second. Typical *events* are the expiration of a timeslice, completion of disk I/O, or completion of terminal input. Events usually affect exactly one process. For example, a timeslice expiration affects only the process running at that time.

Because priorities change at events instead of periodic intervals, an event scheduler can be more responsive than a periodic scheduler. An event scheduler, because it operates on fewer processes at any instant (e.g., about one per event), can be more efficient than a periodic scheduler that adjusts priorities of all processes at some interval.

### 2.1. ESCHED and Our Changes

ESCHED [Straathof86] [Straathof87a] is an event-driven scheduler developed at the University of Maryland as a replacement for the scheduler in 4.2 BSD UNIX. ESCHED rewards or penalizes processes with *boosts* for various activities. Timeslice expiration penalizes a process. Completing terminal input (indicating that this process is more interactive) rewards the process with an improved priority. The process with the best priority is selected to run and loses the CPU only when the process sleeps on an event or a better priority process becomes ready. Timeslice expirations penalize the current process and force the scheduler to select the best priority process again.

We made several algorithmic changes to the ESCHED scheduler. One change was the development and implementation of an algorithm that prevents process starvation; it guarantees that a process will make progress. We also changed ESCHED's basic queuing algorithms to ensure timely response for processes in the kernel. We reimplemented *nice()*, which did not exist in the form we wanted.

In addition to our algorithmic changes, we made several implementation changes to the ESCHED scheduler. We collected the tuning variables used in ESCHED into a single structure and implemented a system call that (along with other scheduling related actions) allows programs to read and modify that structure.

We increased the number of run queues from 32 to 96. We believe that 32 run queues is inadequate for a system supporting thousands of processes. We also increased the priority range from 32 to 98,304 (e.g., 96k). The larger set of run queues and increased priority ranges provide a finer granularity for selecting the appropriate process to run.

While adding more run queues, we examined the process selection algorithm for possible improvement. The original implementation does a linear search through 32 bits of the *whichqs* variable to find the first non-empty run queue. We improved this by adding byte-sized comparisons before the loop to skip large empty sections of the 32 bit value. We expected this change to quarter the select time (we only scan 8 bits instead of 32) but we saw slightly better results. The bonus came from processes that slept in the kernel; their run queue slot is the first position checked within a byte.

### 2.2. Starvation Prevention

Experience has shown that ESCHED exhibits starvation. A set of one or more processes with high priority can starve low priority processes.<sup>1</sup>

For our environment, we want to guarantee that all processes make progress (albeit, perhaps very slowly). To provide this guarantee, we added a periodic sweep of the run queue to examine the waiting processes for those that have been in the run queue since the last pass. These processes are given an *anti-starvation* boost; they eventually accumulate enough priority to get some CPU time. The algorithm uses generation numbers in the process structure to detect process starvation.

<sup>1</sup> George Goble noted this behavior at Purdue on the Gould NP1 systems, which run the ESCHED scheduler.



### 2.3. Timely Kernel Response

When a process awakens after a kernel *sleep()*, ESCHED enqueues it at its previous priority — regardless of what it was doing when it slept.

If the process enters the kernel, locks a resource and then sleeps, it is prudent to give that process a high priority. With a low priority, the process can sit in the run queue for a long time, holding the locked resource and blocking other processes. To resolve this problem, we enqueue processes that sleep in the kernel on high priority resources (as determined by the priority argument to the *sleep()* function) so they will run before any user process. When they wake up, they are dispatched quickly and are allowed to run for a *short time* so they can unlock the kernel resource. They are then rescheduled at their normal user priority. With current test hardware, the *short time* is 10 milliseconds; high resolution timers in the P1 will allow us to use shorter intervals.<sup>2</sup>

Instead of always using the process's base priority, we have a separate queuing priority. When a process is running normally, its queuing priority matches its base priority. Processes that sleep in the kernel adjust their queuing priority to reflect their high priority wakeup requirements.

### 2.4. Nice()

ESCHED dropped *nice()*'s concept as a hint for relative ordering of processes. Instead, relative ordering is done by adjusting the priority bounds of processes.

We wanted *nice* to work similarly to its behavior in the standard UNIX scheduler, where *nice* acts as a skew in the queue priority of a process. We calculate a *nice term* whenever a process's *nice* is changed. This term is included in the queuing priority of a process much the same way as it is in the standard UNIX scheduler.

$$\text{queuing priority} = \text{base priority} + \text{nice term} \quad (1)$$

We modified our starvation prevention algorithm to use a process's *nice* value to scale the size of the boost applied to a starving process. We apply smaller boosts to *niced* processes than we apply to non-*niced* processes. A *niced* process still is guaranteed progress, because we are raising its priority. However, its progress will not be as fast as the non-*niced* process.

## 3. Fair Share Scheduling

Fair share scheduling allows the processor to be apportioned in such a way that different groups of users are allocated a certain percentage of the CPU time. A Fair share scheduler works to ensure that each group receives at least its allocated share of the processor.

Fair Share schedulers work with *sharegroups*. Processes are assigned to sharegroups. Typically, all of a user's processes belong to the same sharegroup. As implied by the name, a certain share of the processor is allocated to each sharegroup. The sharegroup summarizes the allocation of time to that group, the CPU actually used by the group, and other related information used by the scheduler. Within a sharegroup, the scheduler acts as a process scheduler. Between sharegroups, the scheduler works to match the CPU allocation to each sharegroup.

We read the literature describing several existing Fair Share schedulers for UNIX: the FSS scheduler and the SHARE scheduler. FSS, from AT&T, is described in [Henry84]. SHARE is described in [KayLauder88].

<sup>2</sup> One suggested solution to timely response was to set *runrun* immediately when we schedule one of these processes and force the context switch when we leave kernel mode. This prevents the process from running any user code. We would like to let the I/O bound process (e.g., *dd*) execute enough user code to initiate another I/O request and relinquish the processor.

### 3.1. FSS

The FSS scheduler as described in [Henry84] is a modification to the standard UNIX scheduler that controls the distribution of resource to sets of related processes.

The core of the FSS algorithm is the addition of a term in the priority calculation that degrades priority for individual processes as their fair share group uses more of the CPU. The standard UNIX scheduler priority calculation approximates equation 2. FSS augments the basic equation with a *share penalty*. Equation 3 is the result.

$$UNIXpriority = \frac{\text{recent CPU usage}}{\text{real time}} \quad (2)$$

$$FSSpriority = UNIXpriority + \frac{\text{recent fair share group CPU usage}}{\text{real time}} \quad (3)$$

Thus, a process's priority degrades (gets larger) as either that process or other processes in the same sharegroup use more CPU time. Various scaling factors, described in [Henry84], normalize the *share penalty* and other terms in the equation.

One point made in the paper is that UNIX presents a queuing model for scheduling. Like mathematical queuing systems, exact instantaneous matches with the desired behavior can not be guaranteed, but long term behavior approaches the desired values.

### 3.2. SHARE

The SHARE scheduler is described in [KayLauder88]. SHARE is another fair share scheduler based on the standard UNIX scheduler. It traces its roots to the MUSH (Melbourne University SHare Scheduler) scheduler developed at Melbourne University, Australia.

SHARE is oriented towards long term fairness. Its algorithms run at relatively long intervals; updates are done every 4 seconds. Because the fairness interval is long, often measured in days, the maximum error that can accumulate in 4 seconds is relatively small. However, the scheduler takes steps to ensure that short term usage does not exceed the long term allocation by too much. It limits the rate an idle sharegroup can approach its long term allocation, preventing the idle sharegroup from monopolizing the CPU when it becomes active.

Like FSS, SHARE modifies each process's priority with an additional term, a *share penalty*. An algorithmic change from the FSS share penalty in SHARE is that the *share penalty* is scaled by the number of active processes in that sharegroup. SHARE supports hierarchical fair sharegroups, a feature that was not part of the FSS scheduler. SHARE also provides more controls for tuning the scheduler than are described for the FSS scheduler.

### 3.3. Our Initial Fair Share Algorithms

The common denominator in the fair share schedulers we examined was the addition of a new term in the priority calculation that accounts for recent sharegroup usage. We set out to determine the correct way to fit this approach into our event scheduler. Our scheduler already calculates queuing priority as a function of the process's base priority and the nice term. It was not difficult to add a *share penalty* term to the queue priority calculation.

Because we wanted accurate, short term fairness, we decided to record sharegroup usage as a moving average instead of using an exponential decay function. Our fairness interval, called the *grudge period*, is tunable within certain limits. Our default grudge is 15 seconds, recorded as 15 samples spaced 1 second apart. To support different grudge periods, the moving average implementation supports various combinations of sample counts and sampling frequency. One advantage of the moving average over an exponential decay function is that the calculations are exclusively additions and subtractions; no multiplications or divides are required.

We believe in short grudge periods. Our reasoning is that an arrangement for 50% of a machine means 50% at all times, not 100% during the day and 0% at night. The problem with the 100% daytime and 0% nighttime model is that almost everyone prefers to have their 100% during the daytime. We don't think that a user should be penalized during the day for being zealous and using idle cycles during the previous night.

Share penalty calculation involves determining the CPU percentage allocated to each sharegroup and then comparing it with the actual CPU usage of the sharegroup. The group allocation was calculated as shown in equation 4.

$$\text{group allocation} = \frac{\text{group shares}}{\text{all defined shares}} \quad (4)$$

The *defined shares* in equation 4 includes both active and inactive sharegroups. The sharegroup penalty is calculated in equation 5. The resulting penalty contributes to the queuing priority as shown in equation 6.

$$\text{penalty} = \text{scaling factor} * \frac{\text{CPU percentage used}}{\text{CPU percentage allocated to group}} \quad (5)$$

$$\text{queuing priority} = \text{base priority} + \text{nice term} + \text{share penalty} \quad (6)$$

The *scaling factor* converts the fractional CPU usage value into the appropriate units for use as priority adjustments in the run queue. The share penalty scales linearly with the usage. As a sharegroups usage increases, the penalty increases and degrades the resulting queuing priority. The implementation recognizes idle sharegroups to avoid the overhead of multiplications and divides involving zero.

Because we want to maintain interactive response, we place an upper bound on the share penalty; this upper bound insures that the very high priority process in a high-usage sharegroup will run before the CPU-bound process in a low-usage sharegroup. This means that an editor running in a busy sharegroup will usually get good keystroke response because its queuing priority is higher than the CPU bound process in the low-usage sharegroup.

Our fair share algorithm reallocates any unused cycles rather than leaving them idle. The idle cycles are distributed to active sharegroups in proportion to their share allocations. This is a natural fallout from our algorithms and does not require any extra calculations within the scheduler.

#### 4. Scheduling Tools

We developed a set of utilities for the fair share scheduler: administrative tools (defining, deleting, and modifying sharegroups), tools to move processes among sharegroups, and tools to monitor the scheduler.

The *share* utility (not to be confused with the SHARE scheduler) reads a database of sharegroup information. It both defines and removes sharegroups. *Share* has the ability to deal with all or some of the sharegroups defined in the database. Other *share* options allow a sharegroup and its parameters to be specified on the command line.

*Sharegroup* and *resharegroup* move new and existing processes between sharegroups; they work along the same lines as *nice* and *renice*. We have modified *login*, *inetd*, *cron*, *rshd*, and other entry points to the system to place their descendents in an appropriate sharegroup.

*Showgroups* provides a display of the defined sharegroups, process counts (active and sleeping), and CPU allocation and use for each sharegroup. *Showgroups* gives a very understandable picture of who is using the CPU, how much they are getting, and how much they should expect. We find it faster and more obvious to monitor the system with *showgroups* than with *ps -aux*. Included in figure 1 is a sample output from *showgroups*.

---

```

Load average: 1.89, 1.79, 1.67      Grudge:      15 seconds
proc run flag shares (as %) penalty ticks (as %) sharegroup
  11  2          100 14.28% 24575 1030 68.67% kolstad
   6  2          100 14.28% 14620  316 21.07% essick
  12  0   D       100 14.28%  3128   44  2.93% nfs
  10  0   D       100 14.28%   762   40  2.67% default
   9  0          100 14.28%   934   34  2.27% fuller
  34  0  PD       ---      0    34  2.27% system
   4  0          100 14.28%    17    1  0.07% polk
   7  0          100 14.28%    17    1  0.07% shumway
155  4                                1500 100.00% TOTALS (8/27)

```

---

Figure 1.  
Sample *showgroups* Output.

---

## 5. Performance and Tuning

Our fair share scheduler started running in production during November 1988. Since that time, we have monitored its behavior and worked to improve its fairness and responsiveness, as we perceived it. To monitor the scheduler's performance, we used several mechanisms. *Showgroups* was one tool used to see how the system was behaving at a coarse level. We also use the kernel trace facility in the SunOS kernel. By inserting trace points into our kernel, we were able to log the data that the scheduler used to make scheduling decisions, retrieve the log, and evaluate it at our leisure.

With a trace facility in hand, we set about generating scenarios of situations we perceived as being problems for the scheduler. These scenarios involved setting up sharegroups with various allocations and starting appropriate processes in those sharegroups. Some processes were CPU-bound; others cycled between idle and CPU-bound at configurable rates.

Using the tracing facilities and the developed test scenarios, we were able to examine the behavior of the scheduler. In cases where we determined that the scheduler wasn't doing what we wanted, we were able to generate *before* and *after* information to see whether our algorithm changes and implementation changes made any difference. A little work with shell scripts has automated this process to the point where we boot the desired kernel and use a single command to run all of the scenarios, process the data, and produce troff—ready graphs of the results.

### 5.1. Calculating Allocation Percentages

Our initial algorithm for allocated CPU percentage used the number of *defined* sharegroups. Since we have many sharegroups, users were allocated small percentages of the machine; typically 5% or less. Often, only a few of these users were active and they consumed the entire CPU. The resulting very high share penalties were clipped by the algorithm that tries to preserve interactive response for high usage sharegroups.

We changed our CPU allocation algorithm so that instead of using the *defined* shares as a base for calculating percentages, we use *active* shares. This change raised the percentage calculations for the times when a small part of the user community is active and moved the share penalties away from the clipping point. The new allocation calculation formula is shown in equation 7.



$$\text{group allocation} = \frac{\text{group shares}}{\text{all ACTIVE shares}} \quad (7)$$

This matches our preference for short term grudges. We do a better job of equitably allocating the CPU among the users that recently wanted CPU time. Because we've discounted the idle sharegroups (and their unused CPU allocation), we can do a better job of properly allocating the surplus CPU time among the active users.

## 5.2. Incremental Penalties and Smoothing

We believed that our system was reasonably fair, but we didn't like the distribution of cycles. It varied around the allocated percentage more than we believed it should. Trace data showed that the scheduler shifted share penalties such that sharegroups moved into and out of favor at 1 second boundaries. In some cases, a sharegroup would monopolize the CPU for an entire second. This did not match our goal of smooth distribution.

We determined that this was because our update interval (1 second) was a large fraction of our grudge period. The CPU usage changed dramatically across our update intervals. We were only applying corrections every 1/15 (6.6%) of our grudge period. Compare this to the SHARE recalculation every 4 seconds for a grudge measured in hours or days. SHARE applies corrections every 4 seconds; this is 4/3600 (0.1%) for a 1 hour grudge. We changed our share penalty algorithm in two ways to provide a better update rate.

Our first change was to calculate an *incremental share penalty* for each sharegroup when calculating the share penalty.<sup>3</sup> At every clock tick, we add the incremental share penalty to the currently executing sharegroup's share penalty. This provides a reasonable approximation of an instantaneous share penalty. The cycle distribution improved dramatically, but we felt that we could do better.

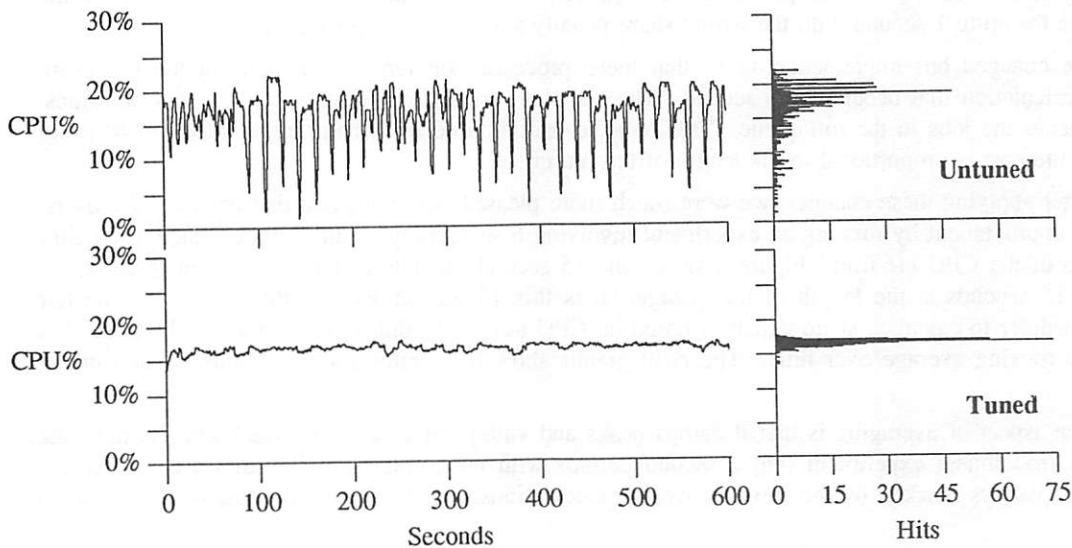


Figure 2.  
15 Second Moving Average Smooth Scenario (16.6% per group)

<sup>3</sup> Our initial effort here cleared the incremental penalty for idle groups. We recently found that it is better to leave the old value in place while a group is idle. This limits the ramp up when a sharegroup becomes active. With cleared incremental penalties, the sharegroup can monopolize the CPU until the next `fss_penalize()` execution (approximately one second).



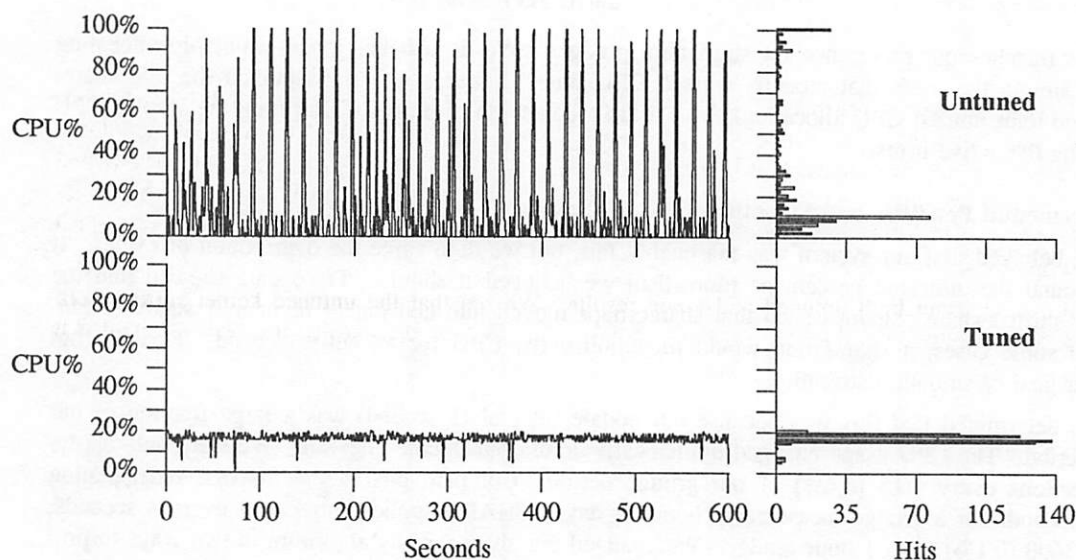


Figure 3.  
1 Second Intervals Smooth Scenario (16.6% per group)

Incremental share penalties provides a better instantaneous share penalty and gives us better queuing priority as process are put into the run queue. However, they do not correct the queuing priority of processes already in the run queue. These processes sit in the run queue, with their old share penalties, until they run or are requeued as part of the anti-starvation sweep. Such processes might remain in the run queue for up to 1 second with the wrong share penalty and the wrong queuing priority.

We changed our implementation so that these processes are requeued as part of the fair share penalty calculation that occurs every second. The fair share routine dequeues, updates queue priorities, and requeues the jobs in the run queue. This requeue operation occurs when `fss_penalize()` runs (every second); the cost is proportional to the length of the run queue.

After applying these changes, we were much more pleased with the cycle distribution. We measured the improvement by running an experiment involving 6 sharegroups with 1 process each, each allocated 1/6 of the CPU (16.6%). Figure 2 shows the 15 second moving average for one of these sharegroups. 15 seconds is the length of the *grudge*. It is this 15 second average that is used by the fair share scheduler to calculate share penalties based on CPU usage. Within figure 2, the graphs to the left show the moving average over time. The right graphs show histograms for each value of the moving average.

One aspect of averaging is that it damps peaks and valleys in a data set. We broke the raw data from the smoothness experiment into 1 second periods with no averaging. This allows us to see the peaks and valleys masked by the moving average calculations. Figure 3 presents this data. It does a

Kernel	15 sec average		1 sec instant		Samples (#secs)
	Mean	Std. Dev.	Mean	Std. Dev.	
UNTUNED	16.5%	3.7%	16.6%	28.9%	601
TUNED	16.6%	0.4%	16.6%	2.2%	602

Figure 4.  
Cycle Distribution of a 16.6% sharegroup.

more dramatic job of exposing rough cycle distribution. Figure 4 summarizes the mean and deviation for the 15 second and 1 second results from both untuned and tuned kernels.

### 5.3. Stability

In addition to a smooth distribution of cycles across sharegroups, we wanted the scheduler to converge quickly on the *correct* set of share penalties and to stay near that value. To test this, we drove the system with a scenario involving four sharegroups. Three of the sharegroups ran CPU-bound processes. The fourth sharegroup ran a process that cycled between CPU-bound and idle. The cyclic process alternated between 90 seconds of idle and 90 seconds of CPU bound real time. This is much larger than our 15 second grudge.

Again, we present both untuned and tuned results. We see that the untuned kernel suffers from poor cycle distribution, but it does converge rapidly to the correct value and then oscillates around that

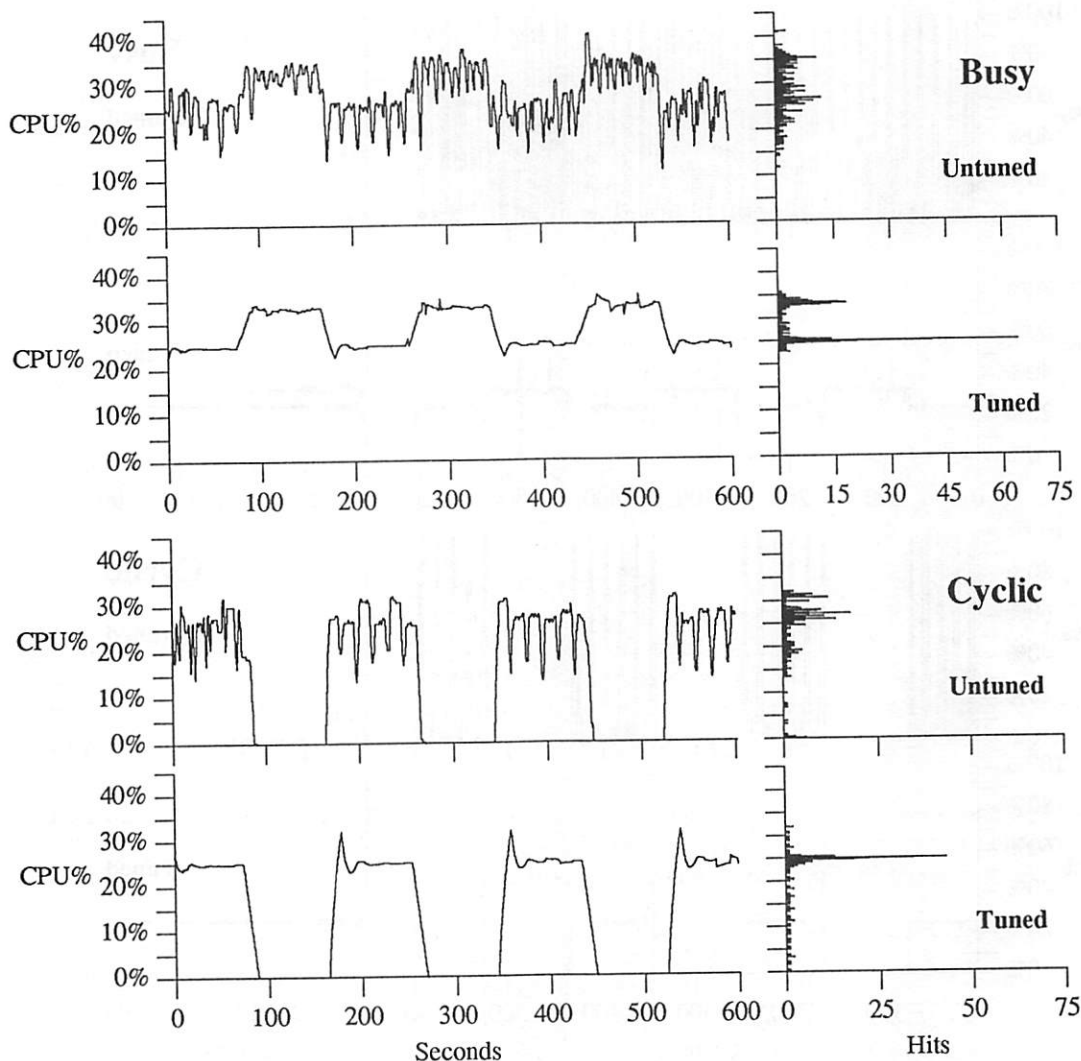


Figure 5  
15 Second Moving Average, Slam Scenario

point. Figure 5 presents information using the 15 second averages; Figure 6 presents the 1 second results. These show us just how well the scheduler is able to maintain a smooth cycle distribution at all phases of the scenario: in equilibrium with both 3 and 4 active sharegroups, during the phase where the cyclic sharegroup becomes active, and during the phase where the cyclic sharegroup goes idle.

The tuned kernel does remarkably well. It quickly converges to the correct value. It does a good job of making sure that the cycles are distributed smoothly once it has converged. We were also very pleased with how the scheduling algorithms prevented the cyclic sharegroup from getting 100% of the CPU during the periods when it first becomes active after being idle.

When the cyclic sharegroup has a period less than the grudge length (a likely case), we saw a different behavior. The scheduler treated the cyclic sharegroup as a group using less than its allocation and gave it a smaller share penalty. The cycles allocated to that sharegroup, but unused by that sharegroup, were redistributed among the other sharegroups according to our plans (e.g., relative to their

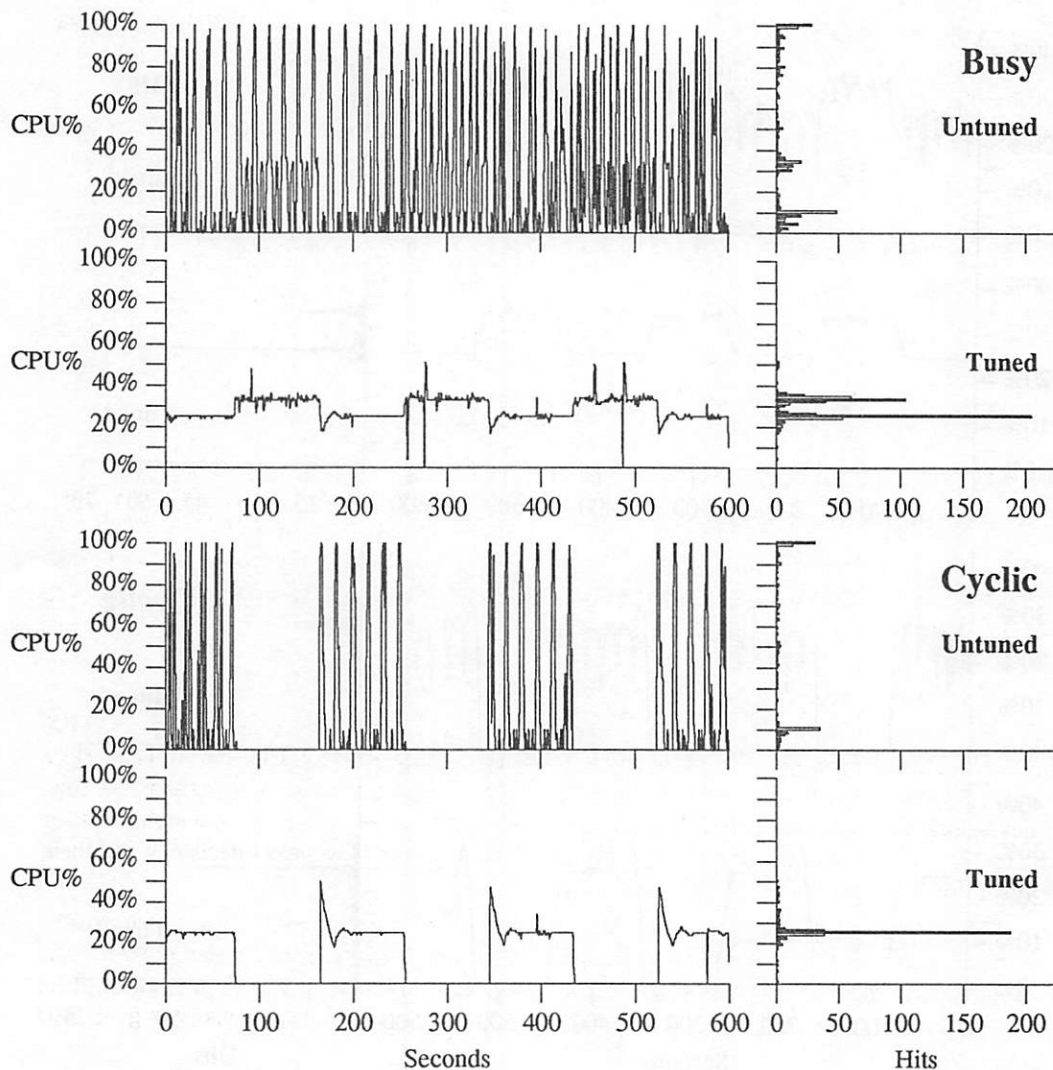


Figure 6.  
1 Second Usage, Slam Scenario

machine share). This made us happy; we would have been less pleased if it turned out that all of our original scheduling algorithms needed modification.

#### 5.4. Nice

Earlier, we discussed how we had to retrofit *nice*() into our scheduler so that a user could impose relative scheduling rankings on his processes. Because of our queuing priority equation,

$$\text{queuing priority} = \text{base priority} + \text{nice term} + \text{share penalty} \quad (8)$$

we had a fair share problem when using *nice*. Two otherwise equal sharegroups, one with a *niced* process and one with a normal process, did not receive equal shares of the machine. The sharegroup with the *niced* process did not receive its fair share; the imbalance is related to the process's *nice* term.

We wanted the effects of *nice* to be isolated to within a sharegroup. A *niced* process, alone in a sharegroup, should get the entire allocation of that sharegroup. The meaning of *nice* can be interpreted as "I don't want it to impact other processes running in this sharegroup" or as "I don't need it now and I'll give up some of my share of the CPU". We can see a desire for both interpretations, but we chose to implement the first interpretation. We view the ability to move processes to other sharegroups with lower allocations as a functional replacement for the second interpretation.

The imbalance results from the *nice* term of the *niced* process skewing the scheduler's attempts to correct any imbalances in CPU usage by changing the share penalty. Our solution was to compensate for the effects of the *nice* term across sharegroups while maintaining its effects within the sharegroup.

To do this, we calculate (as part of the share penalty update) the *least niced active process* in each sharegroup. The least niced process is the one with the smallest *nice* value; a process *nice*(-20) is less than a non-*niced* process, which is less than a process with *nice*(10). We already make sweeps of the run queue to count active processes in each sharegroup and to re-queue processes with their corrected share penalties; we added the *least niced* calculation to this loop. We changed the queue priority equation to use the calculated *nice* base as is shown in equation 9.

$$\text{queuing priority} = \text{base priority} + \text{nice term} - \text{nice base} + \text{share penalty} \quad (9)$$

The *least niced* process in a sharegroup competes equally with against other sharegroups; other *niced* processes in the sharegroup run more slowly. This resolves the problem of the single *niced* process in one group against a non-*niced* process in another group. It also resolved scenarios with processes at negative *nice* values and scenarios with cyclic non-*niced* processes in the same sharegroup as some *niced* processes.

#### 5.5. Billions of Processes Or Paying For Mistakes Up Front

We noticed that two otherwise equal sharegroups, one with a single process and one with 100 processes, did not receive the same amount of CPU time. We determined that this was because we eventually selected those 100 processes and let them run. The scheduler notices (and corrects) its mistake at the next clock tick, but this currently takes 10 milliseconds. The scheduler recovers from the occasional incorrect decision. But this scenario generates a large number of incorrect decisions and their cost (10 ms each) is more than the scheduler can handle.

We changed the fair share calculations so that, in addition to computing the new share penalty and the incremental share penalty, it considers the number of active processes in each share group. The essence of the change is that we *prepenalize* each sharegroup as if we were going to give each active process in that sharegroup 1 tick. This acts as a predictor of which way the share penalty is going and gives us a degree of forward error correction.<sup>4</sup>

<sup>4</sup> The SHARE scheduler does similar things by scaling their penalty for running by the number of processes in that sharegroup.

After installing these changes, we found that the system stayed closer to the fair share allocations. There was still an imbalance in favor of the sharegroup with more processes, but the error was approximately 10% instead of the previously seen 200%.

## 5.6. Hierarchical Sharegroups

As part of our tuning, we looked at implementing hierarchical sharegroups as done in SHARE. Hierarchical sharegroups offer several advantages. A group (e.g., software) may want to allocate its share of the machine across subgroups (e.g., kernel and applications). Also, a hierarchical system allows sibling sharegroups to use any idle time. Continuing with the above example, if the applications group is idle, the kernel group gets all of their time; it is not spread across groups outside of software.

The scheduler changes to implement hierarchical sharegroups are simple. The biggest change is that a sharegroup's CPU allocations changes from what is shown in equation 7 to what is shown here in equation 10.

$$\text{group allocation} = \text{parent allocation} * \frac{\text{group shares}}{\text{ACTIVE shares under parent}} \quad (10)$$

Top level sharegroups have a *parent allocation* of 100% of the processor. We modified the kernel code that installs and deletes sharegroups to maintain the sharegroup lists so that a parent is always processed before its children and we will know the *parent allocation* and *ACTIVE shares under parent* for any sharegroup when we reach that sharegroup.

There were some modifications to the *showgroups* and *share* utilities to handle specification of parent sharegroups and to express the hierarchy. One restriction we placed on the hierarchical implementation is that only *leaf* sharegroups can hold processes. We chose to make this restriction when we failed to arrive at a policy for the relationship of CPU usage between a parent sharegroup and its descendant sharegroups.

One problem we saw in the implementation of hierarchical sharegroups is in the reallocation of cycles when a sharegroup uses only part of its allocation. The scheduler spreads the unused cycles across all other active sharegroups, not just the sharegroups under the same parent sharegroup. This is because the scheduler uses non-zero activity to assume that the sharegroup wants its share. It does not differentiate between the sharegroup that only wants part of its allocation and the sharegroup that wants, but has not received, its full allocation. We have found that our preference for short grudge periods partially alleviates this problem. The short grudge tends to categorize the occasionally active sharegroups as idle instead of active with low usage.

---

```
p = u.u_procp;
p->p_priority += boost[eventcode];
if (p->p_priority > p->p_max)
    p->p_priority = p->p_max;
else if (p->p_priority < p->p_min)
    p->p_priority = p->p_min;
p->p_qpriority = p->p_priority + niceterm - nicebase + sharepenalty;
```

Figure 8.  
Priority Changing code

---



### 5.7. Final Algorithms

The basic event scheduler algorithms are implemented with additive modifications to a process's priority. This code (more correctly, calls that invoke it) is scattered throughout the kernel. Each update involves changing the base priority (an addition), bounding the base priority (a pair of comparisons), and recalculating the queuing priority (several additions). The total cost is the cost per event times the number of events per second. Pseudo-code to handle priority changes is outlined in figure 8.

The starvation prevention algorithm runs at one second intervals; it can be tuned to run at multiples of one second. At that time, it scans the run queue to find processes that have been waiting since the last pass of the algorithm. The algorithm gives each of these starving processes a boost. The algorithm's cost is proportional to the length of the run queue. Pseudo-code for the starvation algorithm is outlined in figure 9. There is code in the context switch routine and in *setrq()* to update the generation number in the process structure.

At clock ticks, we do some work to determine whether we have expired the current quantum. We bump the current sharegroups share penalty by its incremental share penalty. We added a few dozen instructions to each clock tick.

The most expensive operation in the new scheduler is the recalculation of share penalties. We have worked to reduce these costs, but a certain amount of work is unavoidable. We have tried to minimize the expensive multiplication and division operations. Figure 10 outlines the share penalty calculation algorithm. The most expensive section of this algorithm is the calculation of the share penalties; it involves multiplications and divisions. This portion of the code is only performed for active sharegroups. The remaining portions of the algorithm use only addition and subtraction. The iterations of the various loops are either the number of sharegroups (typically smaller than the number of processes in the system) or the number of processes in the run queue (also much smaller than the total number of processes in the system). Some loops, which are combined in the implementation, are separated in the pseudo-code shown in figure 10.

### 6. Further Work

We intend to continue work on the scheduler in several areas. One of these areas is to convert the code so that instead of keeping track of time in *ticks* of 10 milliseconds each, it uses the high precision timers planned for the Prisma machine. We will probably change from *ticks* to either microseconds or nanoseconds. The bulk of this work will be to change affected fields from 32 bit to 64 bit integers.

We also have some processes, such as the NFS daemons, that the scheduler characterizes as interactive processes. As interactive processes, they subvert the share penalty and can take more than their allocation of the processor. We are working on mechanisms to preserve the responsiveness of these processes while throttling their CPU consumption under high loads.

---

```

for (all processes proc in the run queue)
    if (proc→p_generation ≠ master_generation)
        boost(proc);
master_generation++;

```

Figure 9.  
Starvation Prevention algorithm

---

---

```

for (all sharegroups)
    clear leastnice and active proc counters;
for (all processes in the run queue)
    update leastnice marker;
    update active proc counter;
for (all sharegroups)
    update moving average;
    log whether active or idle (for percentage calculations);
for (all sharegroups)
    penalty = scaling_factor * (CPU used / CPU allocated);
    incpenalty = scaling_factor / (clock ticks per CPU allocation);
    penalty += incpenalty * activeprocs;
for (all processes in the run queue)
    re-queue with updated share penalties

```

Figure 10.  
Fair share penalty calculation algorithm

---

We can improve the algorithms used in the hierarchical sharegroup implementation to improve the redistribution of unused cycles. The current scheme redistributed unused cycles across all sharegroups. Instead, we would prefer to see a sharegroups unused cycles distributed to its peer sharegroups under the same parent shargroup.

We have yet to answer the question *did we improve interactive responsiveness*, which is one of our stated goals. The reason for this is that we have neither obtained nor developed a satisfactory test harness that allows us to measure interactive responsiveness. Our impression is that responsiveness is good, but we would like to quantify this measure.

## 7. Conclusions

We defined our reasons and goals for changing the scheduler used in PrismaOS: we wanted to improve system responsiveness for interactive users, we wanted algorithms that would scale well to environments with thousands of processes, we wanted a fair share scheduler, and we wanted to make efficient use of the machine.

We determined that an event scheduler was the proper base scheduler for our purposes and used ESCHED as the starting point for that scheduler. We augmented the algorithms used in ESCHED to get to our desired base scheduler. We then developed and implemented the algorithms needed to put fair share scheduling into our event scheduler.

After running the scheduler in production for several months, we diagnosed several deficiencies in our original algorithms and implementation. For each of these deficiencies, we reasoned out the problems and changed the algorithms to correct the problems. We now have a fair share event-based scheduler that is stable, adapts quickly to changing loads, and provides a smooth service to the sharegroups. We continue to monitor its performance and consider how it might be improved to handle a wider range of user characteristics.

## References

- [Henry84]  
Henry, G.J., "The Fair Share Scheduler", in *Bell Laboratories Technical Journal*, October 1984, Vol. 63 No. 8 Part 2.
- [KayLauder88]  
Kay, J. and P. Lauder, "A Fair Share Scheduler", in *Communications of the ACM*, Volume 31, Number 1, January 1988.
- [Straathof86]  
Straathof, J.H., A.K. Thareja, and A.K. Agrawala, "UNIX Scheduling for Large Systems", in *Proceedings of Denver USENIX Conference*, January 1986.
- [Straathof87a]  
Straathof, J.H., A.K. Thareja, and A.K. Agrawala, "Methodology and Results of Performance Measurements for a New UNIX Scheduler", in *Proceedings of Washington USENIX Conference*, January 1987.
- [Straathof87b]  
Straathof, J.H., "Installing and Operating a New UNIX Scheduler", supplied with ESCHED distribution, 1987.

## Trademarks

UNIX is a trademark of AT&T Bell Laboratories  
SunOS, NFS, and Sun-4 are trademarks of Sun Microsystems, Inc.



**Ray Essick**  
*Prisma*

Ray Essick received all three of his college degrees from the University of Illinois at Urbana-Champaign; the most recent being his PhD in Computer Science in May 1987. During his graduate school years, he dabbled with sendmail, wrote the notesfile system used at some USENET sites, and eventually found time to finish his dissertation on Cross-Architecture Procedure Calls.

Once school was out of the way, Ray went to work for IBM, where he worked on the AIX/370 kernel. Ray worked at IBM for one year before being enticed to come to Prisma and Colorado Springs where they are working on high performance SPARC mainframes. Since August 1988, Ray has been working on schedulers, scaling kernel algorithms, and improving system performance for PrismaOS.



# Parallel STREAMS: a Multi-Processor Implementation

*Arun Garg*

(uunet!sequent!garg)  
Sequent Computer Systems  
15450 SW Koll Parkway  
Beaverton, OR, 97006-6063  
(503) 626-5700

## ABSTRACT

This paper describes Parallel STREAMS, a multiprocessor implementation of STREAMS that has been developed at Sequent. Parallel STREAMS is a key architectural component of the character I/O subsystem of the operating system. Some of the important design issues are discussed. Parallelism in the STREAMS context is explored and the design of a parallel STREAMS scheduler is described. Some of the utilities and guidelines that were added in Parallel STREAMS because of the multiprocessor environment are also described. This paper assumes basic knowledge of STREAMS, although relevant aspects have been summarized briefly in an overview section.

## 1. Introduction

STREAMS is a character I/O mechanism within the UNIX<sup>†</sup> System V kernel that allows the development of kernel drivers (such as networking, ttys, etc) in a modular and uniform manner [Ritchie 84]. It consists of a set of system calls, kernel structures, kernel utilities, and a set of internal interfaces that facilitate a software architecture based on message exchanges between user processes and kernel drivers.

As originally designed, STREAMS is suitable for implementation on conventional monprocessor machines. However, the benefits it provides in terms of modularity, flexibility, ease of implementation, and uniformity become all the more significant when a multiprocessor implementation of the UNIX kernel is considered. This is because of a multiprocessor architecture provides the opportunity of a high performance, *parallel* implementation of STREAMS, in addition to all its other benefits. Such a multiprocessor implementation of the STREAMS mechanism, called Parallel STREAMS, is a key component of Sequent's implementation of the UNIX kernel on the Symmetry<sup>™</sup> platform — a shared memory multiprocessor machine [Gifford 87].

This paper describes some of the more interesting aspects of the design of Parallel STREAMS. To provide some background of the constraints and requirements on the design, the initial sections give an overview of STREAMS and the Symmetry architecture. Following this, the primary goals for the design are described. The rest of the paper discusses the design from the point of view of each component of STREAMS.

<sup>†</sup> UNIX is a trademark of Bell Laboratories.

<sup>™</sup> Symmetry is a trademark of Sequent Computer Systems Inc.



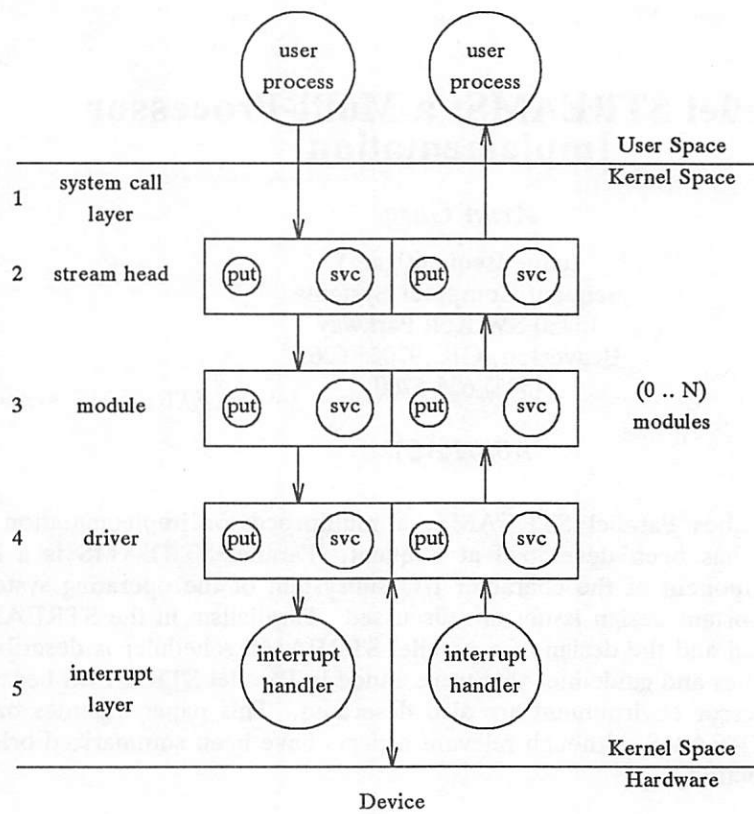


Fig. 1. Schematic view of a stream.

## 2. Overview of Standard STREAMS

This is a brief summary of some aspects of standard (monoprocessor) STREAMS that are relevant to the rest of this paper. The interested reader should refer to [Primer 89] and [Guide 89] for more details of STREAMS.

In the STREAMS context, a user process communicates with a kernel device driver by passing messages on a duplex pipe-like structure called a **stream** (see figure 1). A stream comes into existence after a user process invokes an **open** system call, and consists of a series of STREAMS **modules** interlinked with each other. At the bottom end of the stream is a **driver** module and at the top end of the stream (towards the user process) is a **stream head** module. By means of appropriate **ioctl** system calls, one or more modules can then be inserted between the two end modules of an open stream.

The set of system calls that may be used by user applications to create, use, and destroy streams is — **open**, **close**, **read**, **write**, **ioctl**, **getmsg**, **putmsg**, and **poll**. Associated with each stream module is a pair of queues (a read-side queue and a write-side queue), a set of procedures, and an arbitrary amount of state information. A module's queues are used for the temporary storage of STREAMS messages. All communication on a stream is based on the exchange of messages between the modules in that stream.

The set of procedures associated with the queues of a module are — **open**, **close**, **put**, **service**, and **admin**. To implement a module (say *foo*), a kernel programmer must provide a specific implementation of each of these procedures (*foo\_open*, *foo\_close*, *foo\_read\_put*, *foo\_write\_put*, etc.).

Most of the active message processing within a module is performed by the **put** and the **service** procedures, the other procedures are for housekeeping purposes. The **put** procedure of a queue is invoked at the time a message is first passed to the queue. It processes the message and normally either passes it along to a neighboring queue for further processing, or frees the message. However, if the message cannot be immediately processed for some reason, or if the processing is relatively long, the **put** procedure may defer its processing by queuing the message instead. This is said to **enable** the service procedure of that queue; at some later point in time this service procedure is invoked by the STREAMS scheduler to complete the deferred processing.

All of the kernel processing of the data being exchanged between a STREAMS device driver and a user process may be logically partitioned into the following levels (the numbers 1-5 refer to figure 1):

- 1 System call processing (e.g., data copy between user and kernel space),
- 2 Put/service procedure processing at the stream head queues,
- 3 Put/service procedure processing (possibly none) at the intermediate queues,
- 4 Put/service procedure processing at the driver queues,
- 5 Device interrupt handler processing (possibly none) in the driver.

The processing associated with the system calls (level 1 in figure 1) is done in the context of the user process making the call; i.e., the "thread of execution" is a conventional UNIX process. Similarly, the thread of execution of interrupts (level 5) in the STREAMS context is the same as conventional interrupt processing in UNIX.

However, STREAMS defines the context of execution of the various queue procedures (levels 2, 3, and 4) to be logically separate from a process or an interrupt. The entity that provides the thread of execution of a service procedure is implementation-dependent (e.g., it could be a system daemon process, an interrupt, etc.). The thread of execution of a put procedure depends upon the entity that invokes it (user process, interrupt, or a service procedure). The code for a put/service procedure must, therefore, be designed without assuming any particular process or interrupt context.

The monoprocessor versions of STREAMS have a single, global, FIFO queue of enabled service procedures that is processed by the STREAMS scheduler. Note that this scheduler is logically different from the standard UNIX process scheduler.

As depicted schematically in figure 1, an active stream may be viewed as a duplex pipeline of multiple "threads of execution" between the user process and the device driver (user processes, interrupt handlers, put procedures, and service procedures). In a typical running system there are multiple streams open for the various STREAMS drivers in the kernel, and each stream has multiple, distinct modules pushed onto it.

### 3. Overview of the Symmetry Architecture

The Symmetry Series [Symmetry 87] is a shared-memory, tightly-coupled multiprocessor system that can contain from 2 to 30 Intel 386™ microprocessors with a computational capacity that is scalable up to 120 MIPS. Each processor subsystem has a private cache with hardware support for a **copyback** cache-coherence policy. The system features a fast, pipelined system bus through which main memory, scalable from 4 to 240 Mbytes, can be accessed uniformly by all processors. Similarly, the I/O capabilities of the system are highly scalable and include support for peripherals such as disks, tapes, printers, LAN, asynchronous, and synchronous communication boards, etc.

Important characteristics of the Symmetry architecture are that the processors are all equivalent (there is no master-slave relationship), they share equal access to the main memory, and they all have access to I/O devices. All these components (processors, memory, and I/O) are scalable across a wide range, which makes the system highly flexible.

#### Support for mutual exclusion

The Symmetry hardware supports a **parallel lock** mechanism that is used by the operating system to implement mutual exclusion primitives within the kernel. This mechanism provides an atomic **read/modify/write** operation. The locks are called "parallel" because the system allows multiple *simultaneous* locking operations at the system level by utilizing the caches [Symmetry 87]. Using this underlying parallel lock mechanism, the basic abstractions of a **lock** and a **semaphore** have been implemented within the kernel. A lock provides binary spin-lock functionality for mutual exclusion purposes. A semaphore provides counting semaphore functionality with support for queuing processes waiting for access to the semaphore. It is typically used for mutual exclusion when the contention time involved is too long for spin-locks to be efficient, and for producer-consumer type resource synchronization schemes.

#### Support for interrupt distribution

The Symmetry system uses an independent System Link and Interrupt Controller (SLIC) bus to provide hardware support for dynamic distribution of interrupts among the multiple processors of the system [Beck 87]. The SLIC subsystem supports two kinds of interrupts: a **directed interrupt** to a particular processor, and a **group interrupt** to a group of processors. In addition to I/O devices, each processor can also send both kinds of interrupts on the SLIC bus. Any processor in a group may accept the interrupt; the arbitration is resolved by hardware in favor of the processor running the lowest priority process. This allows the Symmetry system to achieve dynamic balancing of the interrupt load on the system.

#### Support for dynamic load-balancing

Like its predecessor, the Balance Series [Beck 87], Symmetry is a *symmetric* multiprocessor system that makes no distinction amongst its processors. A processor is just another resource that is managed by the operating system. All processors can run application as well as kernel code. Processes may be dispatched on any processor in response to process or system needs. The operating system dynamically balances the total process load among the available processors by maintaining a single, global, priority-based queue of runnable processes, and by maintaining global information about each processor. A process is

™ 386 is a trademark of Intel Corporation.

normally dispatched to run on the processor running the lowest priority process.

The operating system allows multiple system calls to execute *simultaneously* in the kernel, and to access shared data structures. This access is synchronized through the use of locks and semaphores.

#### 4. Design Goals for Parallel STREAMS

It is important to distinguish between the base STREAMS mechanisms and the modules and drivers which utilize the base mechanisms. This paper describes an implementation of STREAMS on the Symmetry system. The design to accomplish this task had to meet the following goals and requirements:

##### Minimize deviation from standard STREAMS

From an application programmer's point of view, STREAMS is a set of system calls with pre-defined semantics described in the System V Interface Definition. It was an absolute requirement that this standard, "black-box" abstraction remain completely unchanged for Parallel STREAMS.

From a kernel programmer's point of view, STREAMS additionally contains a set of internal interfaces, kernel structures, and utilities. It was an important goal of the design to retain as much of this "white-box" abstraction as practically possible. This minimizes the training burden on new kernel programmers, and allows the leveraging of pre-existing STREAMS modules and drivers. This also minimizes the documentation that is needed (beyond that already provided with the standard STREAMS).

##### Maximize performance by parallel execution

One of the primary goals of this design was to allow the drivers and modules to exploit the inherent parallelism present in their streams by mapping it to the parallelism of the underlying Symmetry architecture. This should allow the execution of multiple threads of processing in parallel (on different CPUs at the same time) to obtain an I/O subsystem with high and scalable performance.

##### Provide a simple and easy-to-use STREAMS environment

Writing kernel drivers is quite a complex task on multiprocessors. It was an important goal of the design to make the Parallel STREAMS environment as simple and as easy to use by kernel programmers as possible. It was hoped that a common set of guidelines would emerge out of the underlying design that would facilitate the implementation of various modules and drivers.

#### 5. Parallelism in the STREAMS Context

The standard, monoprocessor version of STREAMS specifies a strict discipline for the execution of service procedures — sequentially, in FIFO order. Since most of the processing within STREAMS is done in the service procedures, a critical question for a parallel design of STREAMS is: "Is this FIFO execution of service procedures necessary or is it merely an



optimization in a monoprocessor environment?" If strict sequentiality is really necessary then parallelism of service procedures is not really possible. This question can be broken down into two separate questions:

1. Is it necessary to guarantee FIFO execution of service procedures belonging to *different* streams? It turns out that there are no dependencies in STREAMS that make FIFO across streams necessary. By breaking this FIFO assumption, **horizontal** or, **inter-stream**, parallelism of service procedures becomes possible — multiple streams can run in parallel with the grain of parallelism being a stream.
2. Is it necessary to guarantee FIFO execution of service procedures belonging to the *same* stream? Again, it does not seem to be necessary, although it seems like a reasonable optimization for a monoprocessor implementation. In a multiprocessor environment, this seems to be an unnecessary restriction. By breaking this assumption, **vertical**, or **intra-stream**, parallelism of service procedures becomes possible — service procedures within the same stream can also execute in parallel.

Parallel STREAMS was designed to realize horizontal, as well as, vertical parallelism. The service procedures associated with *any* two queues within the system can be scheduled for execution completely independently of each other. Recall from figure 1 that the total processing done within a stream may be divided into five different levels. The following sections discuss the design of parallelism from this view-point (the numbers 1-5 refer to the levels shown in figure 1).

### Horizontal Parallelism

Achieving horizontal parallelism of the system call level (1) implies that system calls must be able to execute in parallel. This is relatively straight-forward on the Symmetry machine, because of the design of the operating system's standard process scheduler — recall that the kernel uses locks and semaphores to implement a parallel process scheduler that allows  $n$  user processes to execute  $n$  system calls simultaneously (see [Beck 87] for more details). STREAMS system calls are no different from any other system call — hence their parallel execution does not present any special problems.

Achieving horizontal parallelism of the interrupt level (5) implies that device interrupts must be able to execute in parallel. This was also relatively straight-forward because of the hardware support provided by the Symmetry architecture: recall that the SLIC subsystem provides transparent interrupt arbitration in hardware (see [Beck 87] for more details). STREAMS device interrupts are no different from any device interrupts — hence their parallel execution does not present any special problems.

Thus, all that is *additionally* needed to achieve full horizontal parallelism in STREAMS is the parallel execution of put and service procedures (2, 3, 4). The put and service procedure of a stream must be able to execute simultaneously with those of another stream on a different processor. This has the following implications on the design:

- The STREAMS scheduler must be able to schedule  $n$  service procedures to execute in parallel (i.e., simultaneously and independently on  $n$  different processors).
- To ensure the necessary independence between two streams, the granularity of mutual exclusion locking must be *per stream* or finer. The Parallel STREAMS design is based on a per stream locking model.



## Vertical Parallelism

Vertical parallelism, or parallelism *within* a stream, is achieved to some extent by decoupling system call level processing (1) from the interrupt level processing (5), and from the put/service level processing (2, 3, 4). The synchronization logic has to be carefully designed to retain the natural parallelism between these three levels (e.g., the data copy between user space and kernel space should be done in parallel with internal put/service procedures processing). The only difficult problem that must be solved, before full vertical parallelism is realizable, is that of parallel execution of put and service procedures within a stream. A later section discusses this problem in more detail.

## Parallelism within a Queue

The following questions arise when the issue of parallelism within a *single* queue is examined:

1. Is it important to allow the put procedure to run in parallel with the service procedure of the *same* queue? Put procedures are meant to be quickly processed to reduce the interrupt latencies of the driver. Therefore they cannot be blocked behind a potentially long service procedure. Parallel STREAMS, therefore, allows the put and service procedures of a given queue to execute in parallel. For all the base STREAMS state that is shared between put and service procedures, the mutual exclusion logic is implemented in the base mechanism. However, the module and driver writers are still responsible for the mutual exclusion of their private structures (those that are shared by put and service procedures).
2. Should multiple simultaneous invocations of the put procedure for a given queue be allowed (e.g., by service procedures of two neighboring queues)? This is not really necessary since there is not much benefit to be gained by such parallelism (since put procedures generally do not perform time-consuming processing). However, Parallel STREAMS does allow this parallelism since it simplifies the design of the base mechanism.
3. Should multiple simultaneous invocations of the service procedure for a given queue be allowed? This makes it significantly more complex to design service procedures, and also complicates the design of the scheduler. Parallel STREAMS chose not to allow this; only one instance of the service procedure on a given queue may be active at any point of time. Note that parallel execution of service procedures on *different* queues is still allowed.

## 6. Design of the Parallel STREAMS Scheduler

It is clear from the previous discussion that a key problem that must be solved is that of the parallel execution of service procedures. The Parallel STREAMS scheduler does this in a way that maintains the load-balance of the service procedure among the multiple processors. The scheduler maintains a global queue, `service_Q`, of all the service procedures that are runnable (or, in STREAMS terminology, `qenabled`). The addition and deletion of service procedures from `service_Q` is synchronized by a global lock. However, the actual execution of a service procedure is done by the scheduler *outside* of this lock. This allows, in general,  $n$  instances of the scheduler to execute  $n$  different service procedures in parallel.

The "thread of execution" of a service procedure may be provided by many different entities. Several different mechanisms were prototyped for this purpose; some of these are now described:

- 1 Standard monoproccessor implementations of STREAMS "run" the scheduler (i.e., execute those service procedures that have been enabled to run) from two different places: (i) as part of the processing done during a STREAMS system call (such as write, putmsg, etc.) invoked by the user process, and (ii) during a device interrupt as part of the processing done before returning from the interrupt.

For a multiprocessor architecture, this approach turns out not to be optimal, since it does not always utilize potentially idle processors. While a process is making a write system call on processor 1 (for example) and running *unrelated* service procedures as part of the system call, processors 2 and 3 may be idling. It is better to split the processing associated with a system call (to be done by the user process) from the processing associated with unrelated service procedures (to be done by some other entity) so that these threads of execution may be run in parallel on different processors.

- 2 An approach to solve this problem is to use system daemon processes to provide the thread of execution. A configurable number (say  $n$ ) of system daemons are created at system boot time and act as the  $n$  instances of the scheduler servicing the `service_Q` in a "hungry puppy" fashion. They wait on a common semaphore until a new service procedure is enqueued into `service_Q`. The semaphore is then signaled, which awakens one of the daemons. This daemon removes the first service procedure on the `service_Q` and executes it. It continues in this fashion until `service_Q` is empty, and then goes back to waiting on the semaphore.

This approach couples the STREAMS service procedure load with the load due to system processes. Since, as already explained, the operating system dynamically balances its overall process load among the available processors, the STREAMS service procedure load is also automatically balanced. A disadvantage of this approach is that it involves a context switch for every fresh invocation of the scheduler (i.e., every time a daemon needs to be awakened from the semaphore).

- 3 Another approach to balancing the service procedure load utilizes Symmetry's hardware support for interrupt distribution. As explained earlier, the hardware supports the notion of a group interrupt with an arbitration scheme that is resolved in favor of the processor that is running the lowest priority process. This facility may be used to send a STREAMS scheduling interrupt and an instance of the scheduler is run as the handler for this interrupt. This allows the  $n$ -way parallelism of the scheduler by the simultaneous execution of interrupt-handlers on  $n$  processors. The hardware arbitration of group interrupts ensures that the next invocation of the scheduler is always on the optimal processor and thus dynamic load-balancing of STREAMS service procedures is achieved.

This method is superior to the daemon process approach because it avoids a context switch (on a multiprocessor system, a context switch causes some overhead processing; e.g., the flushing of the translation look-aside buffer).

- 4 The approach that seems to give the best practical results is a variation of the interrupt-based method. As explained earlier in the overview section, the Symmetry architecture has hardware support that allows any processor to send a directed interrupt to any other processor. This facility is used to send a Parallel STREAMS scheduling interrupt and an instance of the scheduler is run as the handler for this interrupt. As before, this allows the  $n$ -way parallelism of the scheduler. Data about the priority of the process running on each processor is kept in a globally shared data structure. This data is used to determine the optimal processor for the purpose of sending the next scheduling interrupt. Thus, dynamic balancing of the STREAMS service procedure load is achieved.

The only real difference between this approach and that based on the group interrupt is that the required arbitration for "the optimal processor to accept the next scheduling interrupt" is done in software instead of by the SLIC bus. The characteristics of the current Symmetry system make it more efficient overall to adopt the directed interrupt approach.

Any interrupt-based scheme must limit the time consumed by the handler to service the interrupt. In a busy system, one can envision a scenario where the `service_Q` is really large — the interrupt that is servicing this queue may run for a long time. An upper bound is maintained on the number of service procedures that are executed by a single invocation of the scheduler. If the `service_Q` is still not empty after the bound is reached, a new instance of the scheduler is "kicked-off" and the current instance ends.

## 7. Design of the System Call Interface

Parallel STREAMS is identical to standard STREAMS as far as the system call interface (open, close, read, write, ioctl, getmsg, putmsg, poll) is concerned. To the application program writer, whose view of STREAMS is limited to this interface, there is *no* difference between Parallel STREAMS and standard STREAMS. This has been verified by running the STREAMS sections of the System V Verification Suite on Parallel STREAMS. To accomplish this, and to add job control logic, the system call layer of STREAMS within the kernel was re-designed using locks and semaphores.

The main problem of designing synchronization logic for system calls is that there are a large number of possible interactions between concurrent calls on the *same* stream. Many of these calls also need to block (sleep) multiple times before they are completed. It becomes quite a complex task to keep track of the synchronization between multiple concurrent calls blocked on different events.

Parallel STREAMS uses locks and semaphores to provide the mutual exclusion needed to accomplish this task. The design has the following features (as any such design might be expected to have):

- Concurrent reads (and, similarly, writes) are not serialized. This is standard Unix semantics for character I/O.
- Reads do not block behind writes and vice-versa. Since a stream is a duplex channel, it would be unacceptable to serialize reads behind writes, or vice-versa.
- All infrequent operations, such as open, close, push, pop, link, unlink, etc., are *serialized* since this made the design simpler without paying too much of a performance penalty.
- The resolution of multiple simultaneous user open and close operations is done at the stream head level so that the module/driver open and close procedure does not have to deal with this level of concurrency. This simplifies the environment from the point of view of a module/driver writer.

The STREAMS ioctl system calls for pushing and popping a module on a given stream, and for linking and unlinking multiplexor drivers [Guide 89] may be called "plumbing" operations, since these operations modify the structure of the stream. These plumbing operations present especially difficult mutual exclusion problems when they are invoked on an "active" stream (in the sense that some of its put and service procedures are running concurrently on other processors).

There are no simple solutions to this problem. Much effort in the Parallel STREAMS design was spent on solving these and similar mutual exclusion problems. Internal primitive routines have been designed to "freeze" a stream during its plumbing and then "unfreeze" it for resuming normal operation. The design is based on per stream locks, semaphores, and reference counts; it is too detailed to be included in this paper.

## 8. Design of the Internal Queue Procedures

The operational view of the module procedures (open, close, put, service, admin) in Parallel STREAMS is the same as that of standard STREAMS. The multiprocessor environment, however, requires a much more precise definition of their semantics because of the possibility of multiple simultaneous invocations. Most of the effort in the design was spent in deriving these definitions (see section 5). Once this was achieved, implementing the routines themselves was relatively straight-forward.

## 9. Design of the Internal Utility Routines

There is a set of internal utilities used by kernel developers of STREAMS drivers and modules. These internal routines encapsulate the operations required by the various STREAMS modules on the STREAMS data structures and provide a data abstraction to the modules and drivers.

To meet the goal of minimizing deviation, all of the standard routines that provide the management of various STREAMS resources (such as **messages**, **queues**, **event-cells**, etc.) have been re-implemented using locks and semaphores in a way that retains their interface. This makes the changes almost transparent to the kernel programmer. This standard set of utilities may be rearranged into logical groups that implement various data abstractions (e.g., a set providing queue-management, another providing message-management, etc.). When viewed in this way, some areas were found to be incomplete. Additional utilities were developed to complete the data abstraction in such cases.

It is important to note that, in the Parallel STREAMS environment, these utilities are the *only* interface to manipulate STREAMS structures, whereas they are just a matter of discipline in the monoprocessor case. Because of this, additional utilities were needed to fill some functionality gaps. Some of these are described in this section.

### Flow control operations

Consider the **canput** utility routine. This routine is invoked to determine the flow control status of a queue and returns true if the queue in question still has room left for at least one more STREAMS message. A typical usage pattern is to call the **canput** routine, and if it returns true, call the **putnext** routine which enqueues a message into the queue that was queried by **canput**.

However, due to the concurrency of events possible in a multiprocessor, the following scenario can arise: The operation **canput** returns true but the queue becomes full before the consequent **putnext** is done. This can happen, for example, because the service procedure associated with the queue (running concurrently on a different processor) may have queued a message in the middle of the two operations. This obviously leads to the high-water-mark of the queue being exceeded by an extra message.



Initially it seems that the only way to prevent such scenarios is by locking out the whole stream during the whole set of operations. This destroys vertical parallelism almost completely. After sufficient analysis, it turns out that the number of extra messages in a queue above its high-water-mark is bounded, because the number of message-producing entities is bounded, and each entity only introduces one extra message. The essential purpose of flow control is to prevent *unbounded* queuing of messages rather than a strict observance of the actual *value* of the high-water-mark. This "bounded-ness" feature can still be preserved in the multiprocessor case without locking the whole stream for the *entire* duration of the set of operations — only during the critical sections.

### The **bufcall** and **unbufcall** routines

The utility **bufcall** is provided by standard STREAMS to recover from the event of a buffer allocation failure; **bufcall** arranges for a driver-specified function to be called when a buffer of a prespecified size becomes available. The standard **bufcall** returns a boolean indicating success or failure. It provides a registration mechanism, similar to that of the familiar kernel utility, **timeout**, to handle buffer exhaustion conditions.

The problem is that standard STREAMS does not provide a de-registration mechanism! This implies that there is no way for a module to cancel a previously registered **bufcall** invocation. This is a serious omission since it makes it impossible for module writers to write code that can gracefully handle the closure of a module or a driver. Consider the following scenario:

1. a stream's service procedure registers a **bufcall**,
2. the stream gets closed and the stream data structures are freed,
3. the **bufcall** finally triggers (after the corresponding stream has been freed),
4. the late **bufcall** then corrupts the data structures of the reallocated stream.

To address these problems, **bufcall** now returns a **token** to identify it for de-registration purposes. A new utility, **unbufcall**, has been added to provide a de-registration mechanism. The **token** returned by **bufcall** is passed as a parameter to **unbufcall** to de-register the call. The scenario described before can now be prevented by invoking **unbufcall** as part of the module's close processing.

Note that the problem just described also exists on monoprocessor environment, and it may be appropriate to adopt a similar solution in standard STREAMS.

### The atomicity of **q\_next** routines

There are several instances in existing standard STREAMS modules of code that directly accesses the linkage pointers of a stream queue to obtain a pointer to the neighboring queue. If *q* is a pointer to the current queue and *q\_next* is a field in the queue data structure that contains a pointer to its neighboring queue, then the code fragments — **canput(q->q\_next)** and **putc1(q->q\_next, ...)** are typical examples.

In a multiprocessor situation, unless the whole stream is locked while the *q->q\_next* pointer is being evaluated, the pointer values may change asynchronously (e.g., when the neighboring module is pushed or popped). Ideally, there should be *no* references to the internal state fields of a queue from a module's put or service routine. Thus new utilities to provide a protected version of this access to a neighboring queue have been added. Examples are:



- The routine `canputnext(q)` implements `canput(q->q_next)` atomically.
- The routine `putnextctl(q, ...)` implements `putctl(q->q_next, ...)` atomically.

#### The `enable_procs` and `disable_procs` routines

The synchronization of the open/close procedures of a module with *concurrently* running put/service procedures of the same module must be designed with great care. The problem lies in the fact that simplistic solutions such as "disallow any active put or service procedure during an open or a close of a module" do not work because there are some modules that require their open/close procedure to exchange messages on the stream. This, in turn, requires that the put/service procedures of the opening/closing module be active for at least some of the duration of the open/close operation.

Parallel STREAMS solves this problem by allowing the module open/close procedures to *control* the point in time at which the corresponding put and service procedures are enabled/disabled. The utility routines `enable_procs` and `disable_procs` have been added for this purpose. These routines may be invoked from a module's open or close procedure (when necessary) to enable/disable the put and service procedures of that module atomically. Any call to the put procedure of a disabled module is transparently "forwarded" to its neighboring module.

### 10. Guidelines for Module and Driver Writers

An important part of the standard STREAMS environment is a set of guidelines that should be followed to facilitate the development of modules and drivers. It should be pointed out that while following the guidelines is good practice even in a monoprocessor situation, it is absolutely *vital* in a multiprocessor situation. During the design of Parallel STREAMS, more guidelines were developed to cover typical multiprocessor situations that cause problems, for example:

- Mutual exclusion problems in the internal queue procedures
- Close procedure cleanup problems (outstanding `timeouts` and `bufcalls`)
- Flow control problems in multiplexing drivers
- Inter-module locking problems

It is beyond the scope of this paper to illustrate all of these situations in detail. These guidelines have evolved over the past year during the development of more than a dozen production quality modules and drivers in the Parallel STREAMS environment. Some of these guidelines are also appropriate for monoprocessor implementations.

### 11. Experience with Parallel STREAMS

Preliminary experience with writing modules and drivers on Parallel STREAMS has been quite positive. At the time of writing, there exist more than a dozen different drivers and modules (tty, pty, console, line-printer, TCP, UDP, IP, ARP, Ethernet, X.25, LAPB, rlogin, telnet, etc.) based on the Parallel STREAMS mechanism.

The Parallel STREAMS environment is so close to the standard one that internal training and documentation has not presented any special problems. The "spirit" of standard STREAMS

has been carried over into the parallel environment. The traditional STREAMS trade-offs (e.g., whether to implement a function in the put procedure or in the service procedure) still apply in the parallel environment. As long as design guidelines are followed, there have been very few surprises, even when writing complex multiplexing drivers (networking engines). "Well designed" modules (i.e., those that obey the standard guidelines, and are well-structured in general) have been easy to port to the Parallel STREAMS environment.

The performance of Parallel STREAMS modules and drivers is still being evaluated. Early experiences indicate that even a single stream can usually saturate its corresponding hardware (tty as well as networking) and hence the system throughput depends on factors other than the STREAMS software. Even on a single stream, because of the vertical parallelism, performance increases as the number of available processors is increased. The actual boost in performance depends upon how much inherent parallelism is present in the particular set of modules and drivers. In the multiple stream case, the performance of drivers scales well with the number of processors available, until system hardware limits are reached. Thus Parallel STREAMS has achieved its goal of being able to provide, in a relatively transparent way, the parallelism inherent in Symmetry hardware to the various drivers and modules. This allows the I/O system, as a whole, to be very scalable and with high performance.

Laboratory configurations of Parallel STREAMS modules and drivers with wide-ranging and stressful networking and tty traffic loads show that the implementation holds up well under stress and seems reliable and robust in general. This has validated the choice of additional utilities and guidelines. Some of these additions are probably good candidates for inclusion into monoprocessor STREAMS implementations as well.

## 12. Summary

This paper has enumerated some of the key design issues that were encountered in the implementation of STREAMS in a multiprocessor environment. Some solutions to these problems have been discussed by examining each component of the STREAMS environment in turn. This paper has also described some of the additional internal utilities and guidelines that have been developed for reducing the complexities of a multiprocessor environment. The design of Parallel STREAMS has achieved its primary goals of being very close to the standard, highly parallel, reliable, and easy-to-use for kernel programmers.

## 13. Acknowledgements

The Parallel STREAMS effort has involved the cooperation of many people at Sequent. The author would particularly like to thank Derek Godfrey for helping with the world of STREAMS, Bob Beck, Bob Kasten, Dilip Ratnam, and Dave Olien for helping with the mysteries of the multiprocessor environment. The first brave kernel programmers to use Parallel STREAMS for "real" modules and drivers included Dave Lennert, Corene Casper, Steve Hemminger, Paul Parenteau, Ken Dove, and Noelan Olson: they provided many detailed comments on the early design and helped make it useful. Brandy Coughlin and Dave Ertel did most of the testing and made many suggestions that improved the design. The documentation was done by Troy Landers. The author would also like to thank Basab Dattaray for many pleasant hours of discussion on this paper. Finally, this project was made possible due to the encouragement, guidance, and "enlightened" management provided by Bob Kasten, Mark Anastas, and Ian Johnstone.

## References

- [Beck 87]  
Beck, B., Kasten, B., and Thakkar, S. S., "VLSI Assist in Building a Multiprocessor", *Proceedings of APLOS-II*, October 1987.
- [Gifford 87]  
Gifford, P. R., "Symmetry: A Shared Memory Multiprocessor with Copy-back Caches", *Proceedings of ICCD87*, October 1987.
- [Guide 89]  
AT&T, "Unix System V/386 Release 3.2 STREAMS Programmer's Guide", 1989.
- [Primer 89]  
AT&T, "Unix System V/386 Release 3.2 STREAMS Primer", 1989.
- [Ritchie 84]  
Ritchie, D. M., "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, October 1984.
- [Symmetry 87]  
Sequent Computer Systems, "Symmetry Technical Summary", 1987.



Arun Garg  
Sequent

Arun Garg is currently leading the OSI Communication Software group at Sequent Computer Systems. He was a staff engineer in Sequent's Operating System group where he completed the Parallel STREAMS project described in this paper. His current interests are in the area of multiprocessor-based design of operating systems and communications. Garg received a B.S. in Electronic Engineering from BHU-IT, India, and an M.S. in Computer Science from SUNY at Stony Brook.

# Implementing Berkeley Sockets in System V Release 4

Ian Vessey      Glenn Skinner  
iv@Eng.Sun.COM      glenn@Eng.Sun.COM

Sun Microsystems Inc.  
Mountain View, California 94043

## ABSTRACT

A major goal of the UNIX® System V, Release 4 (SVR4) development effort has been to unify divergent lines of UNIX development into a single system offering the best features of its progenitors. A closely related goal has been to augment the programming interface to “capture” applications written for other UNIX variants, by making the facilities they depend on available in the unified system. The socket interface is a cornerstone of the 4BSD networking facilities, so its presence in SVR4 is an absolute necessity for application capture.

This paper describes the design of the SVR4 socket implementation, the implementation problems we encountered, and their solutions.

## 1. Introduction

A major goal of the UNIX® System V, Release 4 (SVR4) development effort has been to unify divergent lines of UNIX development into a single system offering the best features of its progenitors. A closely related goal has been to augment the programming interface to “capture” applications written for other UNIX variants, by making the facilities they depend on available in the unified system. The socket interface<sup>[1]</sup> is a cornerstone of the 4BSD networking facilities, so its presence in SVR4 is an absolute necessity for application capture.

Our socket implementation effort faced significant constraints. SVR4 has its own networking architecture, inspired by Dennis Ritchie’s streams work<sup>[2]</sup> and the OSI reference model<sup>[3]</sup>. Compatibility considerations made it impractical to contemplate significant alterations to this framework. Thus we had to implement the socket interface in terms of the native System V networking facilities. This requirement translated into a more detailed set of constraints, discussed further in section 3, that dictated the overall structure of the implementation.

The System V networking architecture provides access to protocol implementations through a stylized message-oriented interface (a *service provider* interface) layered on top of the STREAMS subsystem. A given protocol suite (known as a *transport provider*) is implemented as a collection of STREAMS modules and drivers that communicate with a local client through a set of messages that carry requests to the provider and relay its responses back to the client. User level programs communicate with the protocol implementation by opening a stream associated with the transport provider and issuing STREAMS-related system calls. The stream head converts these to and from messages travelling through the stream.

The socket implementation consists of two components that fit into this framework: **sockmod**, a STREAMS module interposed between the stream head and the transport provider; and **socklib**, a library of user-level functions interposed between the system call interface side of the stream head and the application program.



The remainder of the paper discusses the socket implementation in greater detail, with particular emphasis on the hard parts of the implementation. Section 2 supplies background information on STREAMS and the Transport Provider Interface (TPI) necessary for understanding the rest of the exposition. Section 3 gives an overview of the design and how the constraints influenced it. Section 4 describes the major implementation problems and how we resolved them. Section 5 concludes by giving some insight into the lessons we learned.

## 2. The SVR4 Networking Framework

This section contains a condensed description of the parts of the SVR4 networking architecture most salient to our socket implementation. Readers familiar with STREAMS and the TPI may wish to skip to the next section.

### 2.1 STREAMS Overview

The STREAMS facility is a character I/O subsystem that provides full duplex paths between user processes and devices or pseudo-devices. An individual *stream* is a single instance of such a path. The *stream head* mediates access between user processes and the stream proper, supplying system calls that convert user data to and from *messages* that travel through the stream.

A process can elect to interpose zero or more *modules* between the stream head and the driver. Modules obtain messages from a *queue*, process them in some way, and pass the resulting messages to the queue associated with the next module in line. Since streams are full duplex, each module has two queues, one for the upstream direction (toward the stream head) and one for the downstream direction (away from the stream head). The act of processing a message is the only way a STREAMS module or driver has of affecting its environment; in particular, it has no user context, nor can it directly affect the outcomes of system calls. The implications of this isolation pervade our socket design; a major theme of our design efforts was getting information from a site where it was available to the place it was needed.

Each STREAMS message consists of a sequence of *message blocks*, with each block carrying an embedded type field that identifies its purpose. There are many different types; here we describe only the ones most relevant to the discussion in subsequent sections. The most fundamental message type is M\_DATA, which identifies a message containing application data. M\_PROTO message blocks contain control information such as protocol headers; they often appear as the first block of a message whose subsequent blocks contain related data. M\_IOCTL messages transport data from user *ioctl* requests downstream to the target module or driver, and M\_IOCACK and M\_IOCNAK messages relay responses back to the stream head.

The stream head converts **write** system calls into write-side M\_DATA messages containing the data named in the call and, conversely, satisfies **read** system calls from the M\_DATA message(s) at the front of the stream head's read queue. The **putmsg** and **getmsg** calls behave similarly, but accept and provide *control* buffers as well as data buffers; the control buffer's contents fill and drain M\_PROTO message block prefixes.

### 2.2 Error Handling

Modules and drivers can only communicate with the stream head indirectly, through the messages they send upstream. This isolation can make error reporting difficult. Frequently, the only available option is to send an M\_ERROR or M\_HANGUP message upstream, directing the stream head to reject all future attempts to write data into the stream.<sup>1</sup> Thus, after a write has



caused an error, the module that detects the error can only arrange to make it visible to future write attempts; the true culprit has already completed successfully.

From a module's point of view, reads and writes are asynchronous events, but `ioctl` calls are synchronous. The difference lies in the way the stream head processes them; reads and writes complete after the stream head finishes its local processing, but `ioctls` complete only after the stream head receives an acknowledgement message matching the `M_IOCTL` message it sent downstream to initiate the call's processing. The net effect is that `ioctls` are atomic operations and are the only STREAMS facility that allows implementors to directly couple events occurring in user processes with ones occurring deep within a stream. Our implementation takes heavy advantage of this property, packaging many operations that ostensibly simply read or write data as `ioctls`, thereby allowing it to report errors properly.

### 2.3 TPI

In the System V networking framework, a *transport provider* is a STREAMS driver that implements a transport level protocol such as TCP. A *transport user* is an entity that uses the services of the transport provider to exchange data with a peer.

The Transport Provider Interface (TPI) specifies how a transport service user interacts with a transport provider, using a stylized message-oriented interface. TPI messages arise from applications, which construct them using `putmsg`, or from STREAMS modules, which fabricate them directly. The TPI also specifies a state table that describes all the possible events that may occur at the transport provider interface and the possible states that the transport provider may enter due to an event occurring at the provider interface. TPI defines state tables for both connectionless and connection-oriented transport providers.

The TPI specification is independent of the underlying transport provider and is based on the OSI Transport Service Definition. However, the TPI does not specify local option management and error values returned on a disconnect; instead, these are left up to the individual transport providers to define. Thus transport service users that wish to take full advantage of one of several possible transport providers must be aware of which transport provider is associated with what option management semantics and disconnect error values.

## 3. Socket Design

Our SVR4 socket implementation was governed by two major constraints. The first was:

⇒ The implementation had to be source compatible with existing socket applications.

This was of paramount importance. Existing applications written against the socket interface had to run in an SVR4 environment with no changes. The second major constraint was:

⇒ The implementation had to be consistent with the SVR4 networking architecture.

Socket implementations for previous System V releases<sup>[4] [5]</sup> had shown that these constraints were mutually contradictory. To achieve source compatibility, these predecessors had to resort to

1. The messages differ in how they specify the associated error codes and in their effects on subsequent reads or `getmsg`s; `M_HANGUP` allows pending data to drain with subsequent reads returning EOF, whereas `M_ERROR` forces the stream head to reject all read attempts immediately.

duplicating parts of the STREAMS framework or intercepting system calls. However, we had one advantage over previous implementations: we had the opportunity to make changes to SVR4, subject to some additional constraints. First:

⇒ There could be no changes to the SVR4 kernel specifically for the socket implementation.

Any change to the SVR4 kernel had to be justified for reasons more substantial than the socket implementation's convenience. If a proposed change was considered to be generally useful, then it was acceptable. Moreover, changes had to be backward compatible, so that old applications would continue to work without modification. Two further constraints follow, first:

⇒ There should be no duplication of existing kernel code.

This constraint precludes duplicating code with the aim of changing it. The maintenance problem of having to apply bug fixes to two similar sets of code and having to test each is unacceptable. Moreover:

⇒ There should be no system call wrappers.

We refused to modify the C library to detect invocations of system calls like **read** and **write** on socket descriptors and then behave differently than it would for invocations on other descriptors.

### 3.1 Design Alternatives

We considered several possible designs for our socket implementation.

For instance, there is the socket stream head approach. In this design, socket primitives are either system calls or library calls that map into a pseudo-device driver. This device driver supports the socket interface as well as acting as the stream head. Although this approach manages to emulate socket semantics closely, it explicitly bypasses the SVR4 networking framework by avoiding the stream head. In fact, it replicates the stream head code, contravening another of our constraints, and would effectively result in the maintenance of two stream heads, which we considered unacceptable.

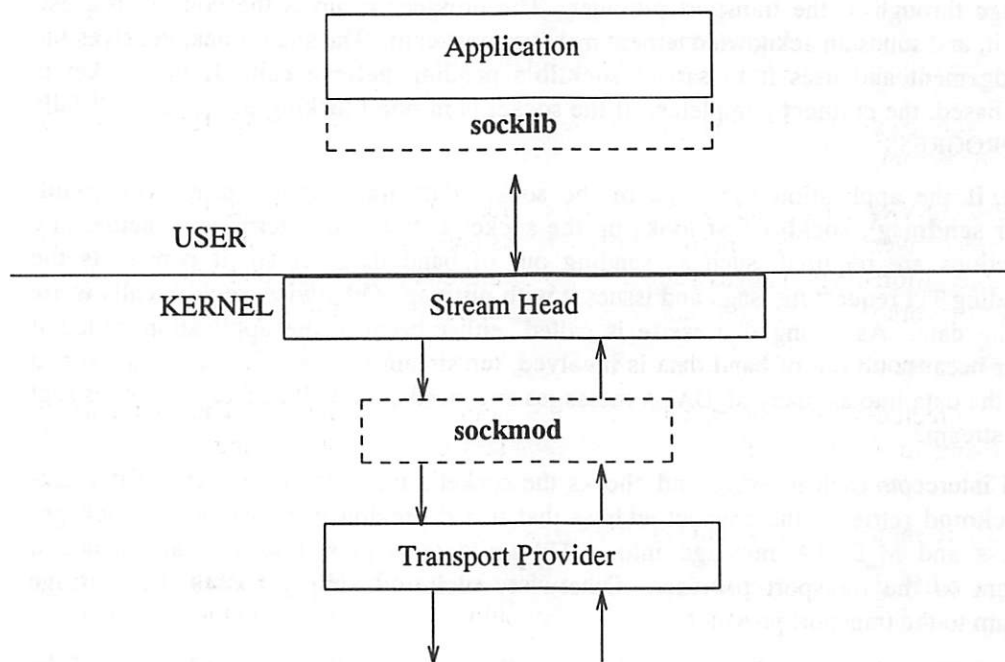
The final design that we considered, consisting of a user library, **socklib**, and a STREAMS module, **sockmod**, interposed between the stream head and the transport provider, was the one that we ultimately adopted. It fit neatly into the framework and there was a precedent for its structure in the Transport Library Interface (TLI) implementation. With the opportunity of changing parts of the framework that were not amenable to our needs, given the constraints outlined above, the choice seemed clear. We knew at the outset that some socket semantics would be hard to attain using this design, but considered that they were of low importance and could be sacrificed if necessary. We also realized that although TLI uses the same architecture as we proposed to use, we might well face issues that TLI never had to.

### 3.2 Implementation Overview

Figure 1 shows our socket implementation's architecture.

#### 3.2.1 The Role of Socklib

**Socklib**'s purpose is to translate socket operations into operations consistent with the SVR4 networking architecture. In doing so it relies heavily on **sockmod**. To understand how the design works, consider the sequence of events that occurs when a typical socket application creates a socket, connects it to a foreign peer, and exchanges data.



**Figure 1.** Socket Architecture

*Creating a socket:* In SVR4, this entails mapping the arguments passed in the socket call to the appropriate device file name using the SVR4 Network Selection Facility<sup>[6]</sup>. Once the device file name is known, **socklib** opens the file and calls **ioctl** to push **sockmod** onto the stream, inserting it between the stream head and the transport provider. **Socklib** then creates and initializes an entry for the socket in an internal list of active sockets before returning to the caller.

*Binding an address:* **Socklib** formats a TPI *bind request* message, packages it into an **ioctl**, and issues it on the stream corresponding to the socket descriptor. The stream head converts the **ioctl** into an M\_IOCTL message and sends it downstream, putting the calling process to sleep pending receipt of the reply message. **Sockmod** intercepts the message, extracts the embedded TPI bind message, updates state information from its contents, and passes it downstream. The transport provider receives the message, processes it, formulates a TPI *bind acknowledgement* message, and sends that message back upstream. **Sockmod** intercepts the message, inserts it into an M\_IOCACK message, and sends the result upstream. The stream head receives the M\_IOCACK, copies its contents to user space, and completes the **ioctl** by awakening the process. **Socklib** examines the bind acknowledgement, verifies that the address requested was the one returned, updates its internal state, and either returns successfully or undoes the bind request (through another interchange with the provider) and returns a suitable failure indication.

*Connecting to a peer:* **Socklib** formats a TPI *connect request* message, packages it as the control part of a **putmsg**, and issues the call. The stream head processes the call and sends the message downstream. **Socklib** then calls **getmsg** to wait for the corresponding TPI *ok acknowledgement* message, blocking until it arrives. Meanwhile, **sockmod** intercepts the connect request message. If the socket is a datagram socket **sockmod** stores the destination address for later use and sends an acknowledgement message upstream. Otherwise, it passes

the message through to the transport provider. The provider receives the connect request, processes it, and sends an acknowledgement message upstream. The stream head receives the acknowledgement and uses it to satisfy **socklib**'s pending **getmsg** call. If the socket is datagram-based, the **connect** completes. If the socket is in non-blocking mode, the call fails with **EINPROGRESS**<sup>2</sup>.

***Sending data:*** If the application used one of the socket data transmission primitives (**send**, **sendto**, or **sendmsg**), **socklib** first looks up the socket's state and determines whether any special actions are required, such as sending out of band data. If so, it constructs the corresponding TPI request message and issues it with **putmsg**. Otherwise, **socklib** calls **write** to send the data. Assuming that **write** is called, either because the application called it directly or because no out of band data is involved, the stream head services the request and packages the data into as many **M\_DATA** messages as it needs, until all of the user data is sent down the stream.

**Sockmod** intercepts each message and checks the socket's type. If the socket is datagram-based, **sockmod** retrieves the connect address that it had previously remembered, packages the address and **M\_DATA** message into a TPI *unitdata request* message, and sends it downstream to the transport provider. Otherwise, **sockmod** simply passes the message downstream to the transport provider.

***Receiving data:*** Since the socket is connected, the application can call either **read** or one of the socket data reception primitives (**recv**, **recvfrom**, or **recvmsg**). If it uses a socket primitive, the library first looks up the socket's state information. If some special action is required, such as receiving out of band data or peeking at unread data, **socklib** issues an **ioctl** to process it. Otherwise it calls **getmsg**, specifying the application's data buffer in the data part and a scratch buffer associated with the socket in the control part.

When the transport provider receives data on the socket, it packages it into either a TPI *data indication* message, if the socket is stream-based, or a *unitdata indication* message, if the socket is datagram-based, and then sends the message upstream. **Sockmod** simply relays the message upstream, where the stream head receives it and copies it out to the control and data buffers specified by the socket library's previous **getmsg** call, which then returns. If the socket is datagram-based and the application wishes to have the address returned, then the library copies the appropriate part of the control buffer into the address buffer supplied by the application. Either way, the library function then returns.

While the foregoing discussion of how typical socket primitives work has given an overview of the general principles of the implementation it has certainly not shown all the details. For instance, it does not describe how implicit binding occurs. Nor does it take into account error handling or issues associated with the library's state management. However, it has illustrated how **socklib** and **sockmod** interact.

### 3.2.2 The Role of Sockmod

**Sockmod** has two purposes, first to mediate **socklib**'s interactions with the transport provider and second to handle aspects of socket semantics that do not map directly into the SVR4 networking

2. The application can determine when the connection has been established by calling **select** and specifying write events.



architecture. The previous section covered **sockmod**'s mediation role. This section concentrates on its role of supporting socket-specific semantics.

*Unix domain addresses:* These are pathnames, which identify the file to be used as the address of a socket. Because pathnames are not canonical, the BSD socket implementation uses the file's inode as the address when routing data and to support bind and connect requests. Our implementation uses a similar scheme, but instead converts a pathname to a device/inode pair, and uses this, together with the pathname, as the address packaged in a subsequent TPI request to the loopback transport provider. To prevent the provider from seeing both parts of the address, **sockmod** intercepts the request and adjusts its address part to ensure that the provider only sees the device/inode pair. **Sockmod** also remembers the pathname part of the address so that it can reply to **getsockname** and **getpeername** requests with a pathname, just as the application would expect.

*Socket-level option processing:* In the BSD implementation, all attempts to set socket-level options succeed, except in the case where one of the arguments to the socket primitives was specified incorrectly. This is true even though some options are intended for other protocol layers, such as the transport layer. For instance, the socket level option **SO\_REUSEADDR** is really meaningful only to the protocol level. In the BSD implementation, these so-called socket-level options are never passed to the protocol layer, but even so, because the protocol layer shares data structures with the socket layer, the protocol code can interrogate them at will. Our implementation has no such shared data structures, and protocol layer options have to be sent to the protocol layer in TPI *option management request* messages. However, not all transport providers will understand the options format used, since this is provider-specific and even if they did they might not support the particular option concerned. In this eventuality they will return an error. Yet socket level options must always succeed. **Sockmod** resolves this conflict by faking success to the application and storing whatever values are necessary to make it appear that the option request was successful.

*Using connect on datagram sockets:* Recall that when the application calls **connect**, a TPI *connect request* message is sent down the stream where **sockmod** intercepts it. If the socket is datagram-based, then **sockmod** remembers the remote address requested in the message, formats a TPI ok acknowledgement message and sends it back upstream<sup>3</sup>. When **sockmod** subsequently intercepts **M\_DATA** messages it formats a unit data request message, inserting the remote address specified in the previous connect, and sends the message downstream to the transport provider.

*Non-blocking connect requests:* If **socklib**'s connect function determines that the socket is in non-blocking mode then it sets a field in the TPI connect request message. When the TPI connect request message has been acknowledged with a *ok acknowledgment* from the transport provider, **sockmod** puts a large enough message on its write queue to saturate the stream's write side. Because of this induced flow control blockage, **select** on write events returns **FALSE**. When **sockmod** eventually intercepts the TPI *connect confirmation* message from the transport provider, it removes the oversized message, which in turn causes selecting processes to receive notification. In the non-blocking case, **sockmod** does not send the

3. In the case of datagram endpoints, there is no interaction with the transport provider for connect requests, because the TPI does not allow the connect operation on connectionless transport providers.



connect confirmation message upstream, but simply discards it. One unfortunate implication of this technique is that if a module is pushed on top of **sockmod**, it must either have no write service procedure, or sacrifice the non-blocking feature. This requirement is unfortunate, but necessary.

This section has sketched **sockmod**'s major tasks. Some of the implementation techniques impose constraints on the behavior of STREAMS modules pushed on top of **sockmod**. Many of **sockmod**'s duties are tasks that other parts of the networking framework really ought to support, and these are discussed in the next section. The fact that the framework does not provide these features means that the socket module is much more complex than it really needs to be. We hope to remove some of this complexity as the networking framework matures.

## 4. Implementation Problems

### 4.1 Use of a user-level library

Initially, one might think that the socket library would be easy to implement. However, it turned out to be much harder than we expected. We encountered two types of problems. The first was: how to maintain state? In contrast to libraries like the C library, the socket library has to maintain state information. Moreover, this state must be distributed, because the socket STREAMS module also has to keep state. Deciding where to keep state information and how to keep it all up to date were key implementation issues. The second type of problem was one of translation: how should the socket library map socket concepts to and from STREAMS concepts? This section describes these problems and their resolution.

#### 4.1.1 Maintaining State

In the BSD socket implementation, the socket primitives are system calls. Since we could not add these primitives to SVR4 as new system calls, the socket library has to map them to the appropriate STREAMS operations. To match the socket primitives to STREAMS primitives, both **socklib** and **sockmod** need to maintain state and other management information. For example, consider the state necessary for handling a **connect** call on a datagram socket. **Sockmod** maintains state that records the connection's status and the corresponding destination address. It is not sufficient for **socklib** to store the address because the application could call **write** to send data on this connected socket, thereby bypassing **socklib**. Both **socklib** and **sockmod** need to record the connection status: **socklib** so that it does not reject **send** calls issued on connected sockets, and **sockmod** so that it can insert the destination address specified as part of the **connect** call into the TPI data transfer messages that it constructs and passes along to the transport provider.

However, there is a problem with **socklib** maintaining state: it can be wiped out when the associated process calls **exec**. It is quite reasonable for socket applications to call **exec** to overlay themselves with a new program that inherits socket descriptors created by the parent program to be used by the new program. For example, when **inetd** invokes **rlogind** the new process is created, and all its external data is initialized. This means that the socket state and management information associated with the inherited sockets is lost. This problem really has two parts. First, how can **socklib** detect that an **exec** has taken place, and second, how can it recover the lost information?

TLI addresses the recovery problem but not the detection problem. An application must discover that it has lost TLI-related state on its own, with no assistance available from any of the TLI primitives. Having done so, it can instruct the TLI library routines to recover their state by

invoking the `t_sync` primitive, which interrogates the transport provider to obtain the missing state information. The socket interface does not include such a primitive and to maintain source compatibility one could not be added. In any case, this kind of solution was undesirable, because it required the application to detect that re-synchronization was necessary.

Our implementation solves the detection problem by relying on the fact that, after an `exec`, the new process's global data is initialized to a known state. Hence, we arrange our initialization of the state information such that the `exec` can be detected. If an application erroneously calls a socket primitive on a non-socket descriptor, `socklib` will see the same state for that descriptor that it would see for a socket descriptor after an `exec`. However, the recovery scheme described below handles this situation properly.

To solve the recovery problem, we capitalize on the fact that `sockmod` is not part of the application's address space. Therefore, `exec` leaves `sockmod` unaffected, so it can act as a repository for `socklib`'s state information. After detecting an `exec`, `socklib` retrieves its state by sending a `synchronization` ioctl to `sockmod`. If it attempts to retrieve state for an invalid or non-socket descriptor, the synchronization ioctl will fail; `socklib` translates the error appropriately before returning to the application. Otherwise, the ioctl acknowledgement contains the lost state and `socklib` initializes its internal structures accordingly. Processing of the original socket primitive then resumes<sup>4</sup>.

For performance reasons, we make `sockmod` keep more state information than it absolutely must. The extra information is actually associated with the transport provider and includes the maximum transport service data unit (TSDU) size, which `socklib` needs, but `sockmod` does not. To prevent `socklib` from having to do two system calls, one to get state information from `sockmod` and one to get the transport provider's information, we have `sockmod` remember both.

#### 4.1.2 Translating Concepts

Translating from socket concepts to STREAMS concepts exposed a different set of problems for the `socklib` implementation to resolve. An example of this type of problem is error checking.

To satisfy the constraint of maintaining source compatibility with existing socket applications, it is necessary to be able to perform the same degree of error checking in the SVR4 socket implementation as in the BSD implementation. The issue here is not whether or not certain error checks should or should not take place but *where* should they take place: in `socklib` or in `sockmod`. It turns out that `sockmod` must do some state checking that `socklib` cannot, and also can accommodate state checking `socklib` could do just as well. For example, it is `sockmod` that must drop any data received after the local application has called `shutdown`, indicating that further receives are to be disallowed, and it is `sockmod` that must support connect requests on datagram sockets. Unfortunately, `sockmod` cannot be left to do all error checking. An application using a socket primitive to transfer data on an unconnected datagram socket is one case where `socklib` has to do all the error checking. Here `socklib` is forced to use the `putmsg` system call to construct the appropriate TPI message. Since `sockmod` has no ability to reject the system call after receiving the message's contents, `socklib` must do the error checking.

4. In SVR4, TLI no longer requires applications to call `t_sync` to recover lost state after an `exec`.

These considerations led to a rule of thumb that we used throughout the **socklib** implementation. If **sockmod** can reject a socket primitive while still providing the correct error semantics, then **socklib** performs only minimal error checking. Otherwise, **socklib** performs all the error checking. In practice, this means that **sockmod** does most of the error checking on socket management primitives, whereas **socklib** handles most of the error checking on socket data transfer primitives.

Another class of translation problems centers around performance. Translating a socket primitive to the appropriate STREAMS primitive can require more than one system call, which immediately puts the SVR4 socket implementation at a disadvantage relative to its BSD counterpart. In some cases the extra system calls can be removed by keeping additional state, but then that state information has to be recoverable after an **exec**.

A final class of translation problems concerns data transmission. There are two ways for an application program to send data over a connected socket. The first is to use one of the socket primitives such as **send**, and the second is to use **write**. Many applications use the latter because they wish to remain ignorant of whether or not they are communicating over a network. However, some applications have to use socket primitives anyway and thus may be more inclined to use a socket primitive to send data. In the BSD implementation, the socket user would expect to get the same performance using either. This feature is obviously desirable in the SVR4 implementation as well.

The socket library has two different system calls available for sending data. One is **write** and the other is the STREAMS **putmsg** system call. If it uses the former, the stream head fragments the data into messages and sends each downstream. Thus only one system call will be needed. If it uses the latter, **socklib** will have to fragment the data and call **putmsg** as many times as is necessary to send the data. On the other hand, if **socklib** uses the former, an individual message making up the data will have no indication that it is related to one logical data block. With the latter this is possible by setting a field in the TPI *transmit data request* message headers that make up the control part of each **putmsg**. So the question is which of the two system calls should **socklib** use? In this case we decided that, in general, greater performance is achieved if the socket library uses **write** and that the usefulness of indicating an association between messages which are part of the same logical message was minimal for most socket applications<sup>5</sup>.

Translating socket concepts to STREAMS concepts is not one-to-one. One has to be careful to choose this translation carefully to achieve the necessary error semantics and optimal performance. For our socket implementation, this translation has meant that state is maintained in the socket library; this produced maintenance problems that had to be solved transparently to the application.

#### 4.2 Implementation Problems Related to the STREAMS Framework

The STREAMS framework provides both the interface between the application process and the stream and the environment in which the protocols operate. As such it is fundamental to the SVR4 networking architecture, which explains its heavy impact on the socket implementation. Although new STREAMS users will in general work within the facilities provided by the STREAMS framework, pre-STREAMS features such as the TTY subsystem and FIFOs need to have

---

5. At the time of writing, this decision still had to be validated.



their semantics met just as the socket interface does now. Thus there is a tradeoff between keeping STREAMS *pure*, keeping it consistent with the philosophy of its originators, and having sufficient facilities to be able to support the semantics of an existing interface. It is not necessarily clear whether the particular problems we encountered constitute a failure of STREAMS to provide a particular feature or whether they simply result from trying to implement the socket interface into a networking environment that it was never designed to fit. The specific problems that we outline here demonstrate the type of problems that we encountered, identifying areas where STREAMS had no support for the desired feature, where the support was not adequate for our needs, or where the STREAMS architecture simply does not accommodate the concepts that the socket interface requires.

#### 4.2.1 Support for SIOCATMARK

In the BSD socket implementation, there is a mechanism to mark a byte in the input data stream. An application can issue an SIOCATMARK ioctl to determine when the marked byte had been reached. In addition, when the application reads data from its socket input queue, it is guaranteed that the marked byte will not be traversed unless it is at the front of the socket input queue. Support for this feature is split between the protocol and the socket layer, with the protocol layer being responsible for setting a count of bytes from the front of the socket input queue to the marked byte, and the socket layer being responsible for the receive semantics of not traversing the marked byte and also for supporting the SIOCATMARK ioctl itself. Prior to SVR4, STREAMS had no similar mechanism.

Since the feature would be generally useful to modules and drivers that might wish to mark a position in the data stream, only the semantics had to be agreed upon. Therefore, a new I\_ATMARK ioctl has been added to the stream head, and the stream head was changed to respect mark boundaries, so that **reads** never cross over a mark. The transport provider indicates the message block to mark by sending it in an *expedited data indication* message, which **sockmod** then marks.

This problem was relatively easy to resolve. The feature was generally useful and caused no compatibility problems.

#### 4.2.2 Use of Read and Write

Under the socket interface, it is perfectly acceptable, in fact normal, for applications to use **read** and **write** on socket descriptors, rather than the equivalent socket data transfer primitives. The advantage that applications gain by using these system calls is network independence. For instance, if the **ps** command has its standard output connected to a socket that is connected to a remote host, then the remote host will receive the command's output, even though **ps** has no knowledge of the underlying network connection.

The problem we outline here concerns the use of **write** on a protocol, such as UDP, that requires data to be transferred in atomic units. If a **write** specifies more than the atomic unit size, then the system call should fail with EMSGSIZE. To understand the problem, it is first necessary to realize that STREAMS was designed with the philosophy that **read** and **write** are pure byte stream interfaces and, as such, should impose no semantics on the application's use of these system calls, irrespective of what the modules and drivers downstream implement or desire. This philosophy extends to disallowing mode flags to change this behavior. When an application calls **write** and specifies more data than can be transferred in a single message, the user data will be fragmented into multiple messages. The byte-stream philosophy precluded a write mode option to support the atomic semantics that we needed, so we were instead forced to exploit an implementation

artifact.

To determine the size of messages that user data should be packaged into, the stream head examines the maximum and minimum packet sizes of the next module downstream. It fragments write requests larger than the maximum packet size. The implementation artifact that we exploit is that, if the minimum packet size is non-zero, the stream head rejects requests to transmit more data than the maximum packet size. In doing so, it incorrectly assumes that the user data cannot be fragmented properly.

Our solution is to set the minimum packet size to one when the protocol requires that data transfer occur in atomic units. Establishing this setting tricks the stream head into the fragmentation behavior we require. Hence, a transport provider with an atomic data requirement should set its write-side minimum packet size to one and its maximum packet size to the atomic data size. When **sockmod** is pushed onto the stream as part of a **socket** call, it peeks at the transport provider's values and sets its own to match. Provided that **sockmod** is the topmost module on the stream, its values will determine how the stream head fragments data.

Although this technique solves the problem, the solution is quite unsatisfactory. "Fixing" the stream head artifact it relies on would compromise the socket interface. Moreover, if another module is pushed on top of **sockmod**, that module must follow the same rules for correct operation.

#### 4.2.3 System Call Failures

The next problem we outline occurs when a socket enters a state in which attempts to read from the socket return one error code but attempts to write return another. For example, when a TCP RESET is received on a socket, subsequent **read** calls on the socket must fail with **ECONNRESET**, whereas **write** calls must fail with **EPIPE**.

The normal way for STREAMS modules to indicate that a stream has entered an error state is to send an **M\_ERROR** message to the stream head, but the pre-SVR4 **M\_ERROR** message was insufficient because it only allowed one error code to be returned for both the read and write cases.

Our solution was to extend the **M\_ERROR** message with an alternate form that contains two data bytes specifying separate read- and write-side error codes. If either of these error codes is zero, then the corresponding system calls (**read** and **getmsg** for the read side and **write** and **putmsg** for the write side) proceed unhindered.

### 4.3 Implementation Problems Associated With The TPI

The TPI is a major part of the SVR4 networking architecture because it defines the acceptable way for transport service users to interact with transport service providers. Our socket implementation proved to be a good test of the TPI because it involved the implementation of a well-used, mature socket interface that had been used with a wide variety of protocols. As one might expect, the implementation exposed some deficiencies in the TPI.

To understand why some of the problems to be outlined in this section exist it is first necessary to understand the difference in the way that the socket layer interacts with the protocol layer in the SVR4 and BSD socket implementations. As we have already said, in the SVR4 socket implementation, the socket layer communicates with the transport provider using a message-based interface specified by the TPI. In the BSD implementation, the socket layer and the protocol layer communicate through a procedural interface structured as a vector of operations defined in the protocol layer that are available for the socket layer to call<sup>[7]</sup>. In some cases the



particular problem to be described in this section is a result of a service being available in the BSD socket-to-protocol interface but absent from its equivalent in SVR4. In other cases the problem is a result of a difference in semantics for superficially similar operations. In almost all cases, the problems identify deficiencies in the TPI that if remedied, would enhance its flexibility and usefulness.

#### 4.3.1 Zero Length Messages

The first problem arises from the fact that the socket interface assumes the ability to send zero length datagrams. The problem is that the TPI specifically says that data request messages on connectionless transports will consist of an `M_PROTO` message block followed by one or more `M_DATA` blocks containing at least one byte of data<sup>[8]</sup>. Fortunately, the ability to send zero length data messages has already been incorporated into the XTI specification<sup>6</sup> and since TLI will conform to the XTI specification, we expect that the TPI will be changed accordingly.<sup>7</sup>

#### 4.3.2 Determining Local and Remote Addresses

The second problem that we encountered involved the `getsockname` and `getpeername` socket primitives, which return the local and remote addresses associated with a socket. In the BSD implementation, the protocol interface provides specific operations to support these socket calls, but TPI defines no similar interface. As a result, the burden falls onto the socket layer to provide the necessary functionality, which it does as follows.

We defined two new ioctls, one to be used in conjunction with `getsockname` and the other with `getpeername`. When an application calls either of these, `socklib` issues the corresponding ioctl, which `sockmod` then interprets, returning the appropriate address to `socklib` in the acknowledgment message. To support `getsockname`, `sockmod` remembers the address that the endpoint is bound to, and to support `getpeername`, `sockmod` remembers the address associated with a connect indication when it is accepted.

Although this approach works for some protocols, it does not work well for the `AF_INET` domain. Consider for example a host with more than one network interface and a server application that chooses to listen for connect indications on all of the host's network interfaces. The server can request this kind of operation by supplying the port to listen on together with the special IP address `INADDR_ANY` in its `bind` request. Socket semantics specify that the local address of the socket returned by a subsequent call to `accept` should indicate the same port number as the server was listening on, but because the client bound to an IP address of `INADDR_ANY`, the IP address of the new socket should be the network interface on which the connect indication actually arrived.

The problem is that `sockmod` relies on the `bind` request fully specifying the local address, which it uses to satisfy `getsockname` requests on the socket returned by `accept`. This is a problem for some applications such as the FTP daemon, and to solve it we added the ioctls used in conjunction with `getsockname` and `getpeername` to the `AF_INET` transport providers themselves; `socklib` will only service these ioctls if the particular transport provider does not. Since the transport providers know the real address to associate with the accepted socket there is no problem.

6. See `t_sndudata`, x/Open Portability Guide - Network Services

7. In SVR4, TLI allows an application to send data messages of zero length provided the underlying transport provider supports it.

It is very unsatisfactory to have to add these `ioctl`s to the transport provider when the real answer is that the TPI should include in its interface specification explicit support for returning this kind of information at connect time. We are pursuing this change to the TPI.

#### 4.3.3 Option Management

The third TPI-related problem we encountered concerns local option management. The socket interface allows local option management to be done at any time, whereas the TPI specifies that it can only be done after an endpoint is bound but before it is connected. In TPI terminology, local options can only be negotiated when the endpoint is in the `T_IDLE` state.

The socket interface not only allows options management at any time, but in some cases, a socket option has to be in effect before the socket is bound to achieve the desired action. For example, the `SO_REUSEADDR` option must be in effect before the endpoint is bound because it affects the semantics of the bind operation itself. The BSD socket implementation does not suffer from this problem, partly because it does not have any such state requirements, but also because the socket layer and the protocol layer share common data structures. This sharing means that some options don't even have to be passed to the protocol layer, even though it is the protocol layer that interprets them. In fact, the `SO_REUSEADDR` option is a socket layer option, and is never given directly to the protocol; instead the protocol examines it at bind time through the shared data structures.

Since the endpoint's state is not part of the criteria that the `AF_INET` transport providers use to validate socket level option requests, the solution we employed was to remove the state check altogether. Again, we do not consider this solution to be satisfactory because it means that the implementation does not strictly conform to the SVR4 networking architecture, specifically the TPI. The only satisfactory answer is to remedy the deficiency in the TPI, and we are trying to do so.

#### 4.3.4 Bind/Listen Semantic Differences

The problem to be outlined here results from differences between the socket interface and TPI on superficially similar operations. The typical sequence of events for a socket-based server is to create a socket, bind it to a well-known address, call `listen` to set the maximum number of unaccepted connect indications (backlog) that the protocol must enqueue, and then call `accept` to wait for connect indications to arrive. However, the TPI specifies the backlog value in the bind request rather than in the listen request. So a socket application expects to specify the backlog in `listen`, whereas TPI requires the backlog to be specified in `bind`.

Our solution to this discrepancy relies on the ability to unbind an endpoint after it has first been bound. When an application calls `bind`, `socklib` specifies a backlog of zero to the transport provider. (A value of zero is normal for applications that do not wish to receive connect indications.) If the application subsequently calls `listen` with a non-zero backlog value then the request translates to an `ioctl` that `sockmod` services. When `sockmod` receives the `ioctl` message, it first unbinds the endpoint, and then rebinds it to the same address, specifying the new backlog value. The fact that in some cases the endpoint has to be unbound and then rebound introduces a window in which another application could request the same address. However, this window is small and we consider it acceptable.

#### 4.3.5 Accepting connections

The final problem in this section is related to the local address assigned to the socket returned by `accept`. To understand why the problem exists, it is necessary to understand how the operations

underlying an **accept** in the BSD implementation differ from their TPI counterparts.

In the BSD socket implementation, a call to **accept** causes the socket layer to allocate a new socket descriptor, which it then passes to the per-protocol accept routine. In the internet domain, the accept routine associates this new descriptor with the socket that originated the connect indication and assigns its local address. Thus, it is the protocol layer that determines the accepted socket's address.

In the SVR4 networking architecture, the **accept** call translates to a TPI *connect response* message<sup>8</sup> which **socklib** generates and sends to the transport provider on the accepting socket. One component of this message is an open, *bound* file descriptor corresponding to the transport provider and on which the new connection will be accepted.

The problem is the fact that the file descriptor used in the connect response message must be bound. For some protocols, this may be acceptable, but in certain cases, only the protocol knows the proper address that should be assigned. Consider the example of a server binding to the IP address `INADDR_ANY`. In this case, the protocol assigns an IP address corresponding to the actual network interface that the connect indication arrives on. The application has no idea what this IP address is. Only the protocol layer knows it, so only the protocol layer can set the accepted address correctly.

We believe that the solution to this problem is for TPI to allow the endpoint specified in the connect response message not to be bound. If the application chooses not to bind it, then the transport provider should. This change allows the application or the protocol to decide the issue.

## 5. Conclusions

This paper has described the design of our socket implementation and the problems that we had to overcome while developing it.

The constraints imposed on the implementation were stringent, but reasonable; it was the SVR4 networking framework that gave us trouble. Our task in this implementation has really been to try to integrate the socket library, stream head, socket module, and transport provider into a unified whole whose components cooperate to emulate socket semantics. The BSD socket implementation was designed to work in that kind of environment, with the advantage that its structure closely matched the socket interface. STREAMS was never advertised as providing this kind of tight knit interaction. Quite to the contrary, it supports discrete processing elements, operating independently of each other, with well-defined interfaces, and driven solely by their inputs. Moreover, the socket interface is not without warts; in particular, the socket layer and the underlying protocols are more incestuous than they really ought to be. Therefore, it would have been naive to expect total compatibility between our socket implementation and its BSD counterpart, and some minor incompatibilities still remain<sup>[9]</sup>.

Many of the problems we faced still need acceptable resolutions, especially the ones concerning the TPI. These TPI deficiencies must be rectified if TPI is to be as useful as it should be, rather

---

8. The connect response message is actually sent as an `I_FDINSERT` `ioctl`. The socket library creates a file descriptor suitable for the transport and passes it as part of the `ioctl`'s data. The stream head translates the file descriptor into a STREAMS queue pointer, creates a message from other arguments passed in the `ioctl`, and sends it downstream to the transport provider.

than a hindrance. We also hope to solve many of the problems associated with the STREAMS framework more elegantly as STREAMS matures.

Our socket implementation has shown that it is possible to work within the SVR4 networking environment and still provide a socket emulation that closely follows its BSD counterpart. Implementing sockets within the SVR4 networking framework was not easy. There were many compromises and hard questions of where functionality should go. In many ways, it is a tribute to STREAMS and TPI that the socket implementation is so nearly complete, considering the constraints it had to meet.

All of the standard BSD networking utilities run unchanged on our socket implementation. The X11/NeWS server has been ported to SVR4 without change to any of its socket-related code. These successes indicate not only that our implementation provides all of the significant features of its progenitor, but also that we have met our principal goal of application capture.

## 6. Acknowledgements

We would like to thank Bob Gilligan and Bill Shannon at Sun together with Stephen Rago at AT&T for always being willing to discuss issues that we encountered during this project and for helping to solve many of the difficult ones.



## REFERENCES

1. Joy, William, Robert Fabry, Samuel Leffler, M. Kirk McKusick, and Michael Karels. "Berkeley Software Architecture Manual 4.3BSD Edition", Department of Electrical Engineering and Computer Science. University of California, Berkeley, California 94720, April, 1986.
2. Ritchie, Dennis, "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2, October, 1984.
3. "Information Processing Systems - Open Systems Interconnection - Transport Service Definition", ISO IS 8072, 1984.
4. Lachman Associates, Inc., "System V STREAMS TCP, Release 3.0, Network Programmer's Reference Manual", 1901 N. Naper Boulevard, Naperville, Illinois, 60563-8895, 1988.
5. Wollongong Group, Inc., The, "WIN/386 Reference Manual", 1129 San Antonio Road, Palo Alto, California 94303, December, 1987.
6. *UNIX System V Release 4 Network Programmer's Guide*, AT&T, 1989.
7. Leffler, Samuel, William Joy, Robert Fabry, Michael Karels. "Networking Implementation Notes 4.3BSD Edition", Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April, 1986.
8. *UNIX System V Release 3.0 Kernel Interfaces*, AT&T, 1983.
9. *UNIX System V Release 4 Network Programmer's Guide*, AT&T, 1989.



Ian Vessey  
Sun

Ian Vessey graduated from the University of Keele, Staffordshire, England in 1979 with a joint honors degree in Electronics and Mathematics. After working on the design of quality control systems he became interested in networking and protocols. Ian worked on network management at AT&T for three years before joining Sun where he is currently a member of the Internetworking group.



Glenn Skinner  
Sun

Glenn Skinner first discovered UNIX in 1978 when a shiny new PDP-11/60 arrived at Cornell's Computer Science Department. He's been working with UNIX ever since, first at Ford Aerospace, where he helped write a UNIX emulator for the KSOS secure operating system project, and later at National Semiconductor, where he helped develop the GENIX 4.1 and 4.2 ports. He's been at Sun Microsystems since 1985, participating in many projects, including joint development work with AT&T on System V, Release 4.





## **Two Network Management Tools (How Many Packets Would a Packet Router Route if a Packet Router Could Route Packets?)**

*Allan Leinwand*

Hewlett-Packard  
Corporate Telecommunications  
allan@hpcta.corp.hp.com

*Jeff Okamoto*

Hewlett-Packard  
Corporate Computing Center  
okamoto@hpccc.corp.hp.com

### **Background**

In 1986, the Hewlett-Packard Company (HP) installed its own TCP/IP network, called the HP NET Internet. As of this writing, the number of systems nears 20,000, linked by over 200 cisco Systems routers (specifically cisco Systems Advanced Gateway Servers). The HP NET Internet encompasses the continental United States, England, France, West Germany, Switzerland, Japan, Australia, Singapore, and other locations in the Far East, Latin America and Canada. Approximately 150 - 180 Gigabytes of data per month is transferred via the HP NET Internet.

### **The Problem**

Managing a TCP/IP network of this size is a monumental task. Only recently have tools become available to assist a network administrator. And, at the LAN level, no tools exist that can monitor the status of local hosts and routers, other than by waiting for a catastrophic failure.

At Corporate Telecommunications, the network engineers were continually performing routine tasks and network debugging by hand, interactively querying the routers to determine its status and that of its interfaces.

The Corporate Computing Center has the responsibility to manage many of the company's central computers. If a machine goes down, or the network is experiencing difficulties, the operators needed to be alerted as soon as possible so that they can begin to take corrective action, or to notify the system administrators.

We began independently working on solutions to our respective problems. From the beginning we have assisted each other with advice and coding solutions. The result of this cooperation is a set of tools that can help network engineers do their jobs in a more effective way.

## Software Requirements

The two tools described here both run under HP-UX<sup>1</sup>, HP's version of UNIX<sup>2</sup>. At least version 6.2 for the Series 300 or version 3.0 for the Series 800 is required. Both were written using the X11 HP Widget set. Since both tools rely heavily on color to impart information, a color monitor is strongly recommended.

### Xnetmon: Author, Jeff Okamoto

Xnetmon is a tool to monitor connectivity and network delay to various hosts on a LAN/WAN. It uses the Internet Control Message Protocol (ICMP) Echo function to "ping" specified hosts and routers and display the results on a grid of labels. These nodes are defined in a configuration file. Nodes can be specified by either host name or IP address. Additional information can be added to a label, and blank and text labels can be included to assist in formatting the grid.

After "pinging" the nodes in the grid, xnetmon will change the colors of the labels to reflect the status of the node. The following table explains the main colors<sup>3</sup> used by xnetmon.

Color	Meaning
Green	Node returned all packets.
Yellow	Node returned only some packets.
Red	Node returned no packets.
Black	An error occurred. This could include network errors or a host name lookup failure.
Gray	The node has at some past time not returned any packets, but is returning them now.
Orange	Node is currently being ignored. See text for more details.
Magenta	Initial color of label, or after the node is no longer being ignored.

At user-specified intervals, xnetmon will again "ping" all the nodes in the grid and update the colors. To manually recheck connectivity, simply select the desired nodes (which causes them to change color slightly to show that the node has been selected), then push the "Recheck" button.

Xnetmon will wait for a user-specified amount of time before deciding that a node has not returned any packets. If there are many nodes on the grid, the total time required to wait for xnetmon to complete the "pinging" could become excessive. If the user does not wish xnetmon to "ping" certain nodes, those nodes may be selected

<sup>1</sup> HP-UX is a registered trademark of Hewlett-Packard Company.

<sup>2</sup> UNIX is a registered trademark of AT&T in the USA and other countries.

<sup>3</sup> If you only have a monochrome monitor, or don't like color, xnetmon is capable of using different background patterns to represent the various statuses.

and then the "Down" button is pressed. Xnetmon will change the label's color to orange and will no longer "ping" any of these nodes until they are re-selected and the "Down" button is pressed again. Once this occurs, the labels will be colored magenta.

In addition to checking if a node has returned the "ping" packets, xnetmon also records the average round-trip time of the node. If a node is selected and the "Info" button pressed, an window is displayed that contains information about the node. This information includes the percentage of time the node has returned all, some, or no packets, and the round-trip time averaged over various intervals.

Recheck	Info	XNetmon (1.7)			Down	Quit
Host_1 15.60.0.1	Host_6 15.60.0.6	Host_11 15.60.0.11	Host_16 15.60.0.16	Host_21 15.60.0.21		
Host_2 15.60.0.2	Host_7 15.60.0.7	Host_12 15.60.0.12	Host_17 15.60.0.17	Host_22 15.60.0.22		
This_is_a message_square!	Host_8 15.60.0.8	Host_13 15.60.0.13	Host_18 15.60.0.18	Host_23 15.60.0.23		
Host_4 15.60.0.4	Now_a blank_square		Host_19 15.60.0.19	Host_24 15.60.0.24		
Host_5 15.60.0.5	Host_10 15.60.0.10	Host_15 15.60.0.15	Host_20 15.60.0.20	Host_25 15.60.0.25		

Figure 1: Xnetmon's main display

Xnetmon was successfully used at the Network Operations Center booth at the March 1989 UniForum. Many times the network operators were able to detect network problems and dispatch troubleshooters to the vendor's booth before the vendors were aware of the trouble. Many passersby expressed interest in the program.

Xnetmon was also used to assist in the recovery of the HP NET Internet after the Loma Prieta earthquake of October 17, 1989. Users with a need to transfer files to our intercontinental sites were continually notified of the state of the network and the chances of successfully transferring their files.

Information for hpycia (Japan) Done			
Traversals:	none	some	all
	4	26	524
Packets:	received	sent	percent
	2717	2770	98.09
Running averages in ms for the last traversals			
	5	241.00	
	50	243.12	
	100	244.90	
	200	245.77	
Program startup time: Mon Nov 13 14:44:21 1989			
Returned all packets since: Mon Nov 13 17:21:44 1989			

Figure 2: Xnetmon's information display

### Future work on Xnetmon

Two improvements to xnetmon are being worked on. The first is the decoupling of the "pinging" section from the window displayer. Currently, while xnetmon is "pinging", the program appears not to respond to any X events, such as button and

exposure events. (In reality, these requests are queued and are processed once the program restores control to the main X Toolkit event loop.) Decoupling the two sections would allow the window displayer to respond to events while the other section is "pinging."

The second improvement would be to allow a user to program xnetmon to take arbitrary actions based on user-defined criteria. Examples of these criteria include: if any host is down; if a host with an IP address of 15.60.32.\* is down; if any host has not returned any packets for longer than one hour; if the average round-trip time to any host exceeds 1000 ms.

An action will normally be to execute a program with arbitrary arguments and input. A typical action might be to send mail to the system administrator, or to send a message to a select group of people. In an earlier version of xnetmon, an engineer rigged his workstation to control a bank of lights. When xnetmon determined that a host was down, it executed a program that triggered the appropriate light, thereby notifying the engineer that the system was no longer responding.

The language to be used is based on SPIN (Simple Programmable Interface), which was developed by Jim McBeath of Silicon Compilers Systems Corporation and posted to comp.sources.unix in March 1989 in Volume 18, Issues 9 and 10. It is hoped that greater control will be achieved over xnetmon as it performs various actions.

### **Xpath: Author, Allan Leinwand**

Xpath was first developed to find the route between two IP addresses. It has since grown into a tool that assists network engineers in managing networks of any size, from LAN's to the entire HP NET Internet. Throughout the development process, many network debugging techniques have been automated by xpath.

Xpath gathers its information by opening a session on the router, then parsing the information sent by the router in response to xpath's commands. Naturally, xpath understands the router's command set and in what format it should expect the router's reply. This retrieval of information has proven to be a reliable and efficient method for data collection. Also, xpath employs the use of the ICMP Echo to "ping" various network elements including hosts and gateways.

### **Functionality**

The functionality of xpath can be broken down into six major areas, accessed through pulldown menus. These areas involve network debugging, network monitoring, and network analysis for current usage statistics and possible areas of growth.

On startup, xpath displays a large window with a map of the world. Every HP NET Internet site is represented on the world map with a label. The locations of the labels on the map correspond to the site's geographical location, though not completely to scale. Selecting these labels will open a window containing a list of all the routers registered at that site.

#### **1. Path Functions**

Perhaps the most common user complaints concern network connectivity and network performance. The first area of xpath's functionality specifically addresses



these problems by reporting and displaying the route between any two IP addresses and the delays between the routers that are routing the packet.

After selecting the source and destination, or providing their host names or IP addresses, xpath queries the respective routers between the source and destination.

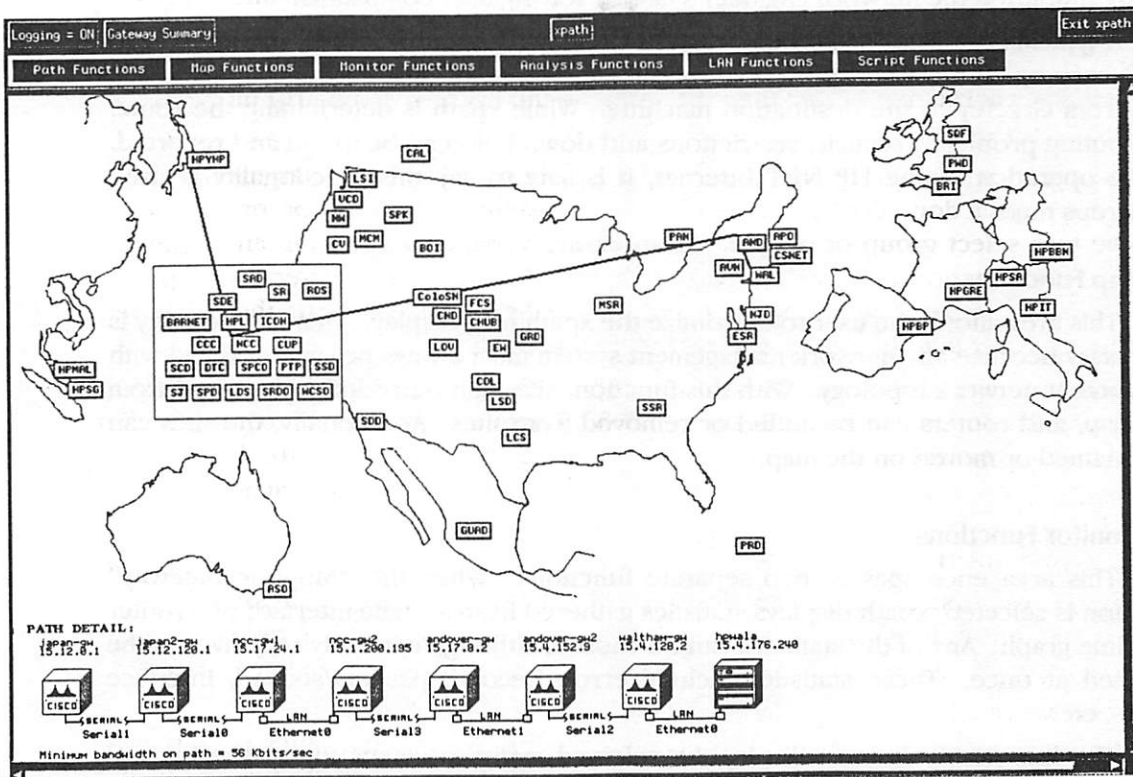


Figure 3: Xpath's Show Path display

The route is displayed on the world map and on a separate section of the main window. The types of routers along the path are displayed in this separate section. On the HP NET Internet, these are either a cisco router or HP 9000 system. The type and speed of connection between each router is displayed (e.g., 9600 baud serial line, 10 Mb/s Ethernet, etc). The minimum bandwidth of the entire connection is also displayed.

If xpath cannot find a path between the source and destination, it displays as much of the path as it can determine and reports any error messages received while attempting to reach the destination.

Once such a route has been found, the network engineer can have xpath analyze the route. Xpath will then "ping" between each of the routers on the path and show the packets turnaround time and success rate, which can help to isolate bad router connections.

The algorithm used in finding the path from the source to the destination IP address is simple. Starting with the closest router, xpath queries its routing tables to find the best known route to the destination IP address. Once this IP address has been obtained, the next router along the path is asked the same question. This process

continues until a router is found that is directly connected to the destination subnet or IP address, or until an error occurs.

This portion of xpath alone has solved numerous network problems such as connectivity problems, performance problems, routing loops, and others. Xpath has greatly simplified the network engineer's task in solving user complaints.

A typical complaint might be: "I am in Greeley (Colorado) and I cannot telnet to hpcta.corp.hp.com". The network engineer can use xpath and immediately find the path from Greeley to the destination machine. While xpath is determining the route, any routing problems, security restrictions and down links can be found and reported. In the operation of the HP NET Internet, it is safe to say this functionality is used numerous times a day.

## 2. Map Functions

This area allows the user to customize the xpath map display. This functionality is necessary because any network management system must always be synchronized with the current network topology. With this function, sites can be added or removed from the map, and routers can be added or removed from sites. Additionally, the sites can be renamed or moved on the map.

## 3. Monitor Functions

This area encompasses two separate functions. When the "Monitor Gateway" function is selected, xpath displays statistics gathered from a single interface of a router on a line graph. Any of the statistics can be chosen, although currently only five can be graphed at once. These statistics include errors/second, packets/second, interface resets, etc.

The line graph drawn will plot the selected statistics versus time. The interval between router queries is user-controllable. If desired, alarms can be triggered if the rate of any of the selected statistics exceeds a user-selected rate. For example, the user may have the system bell rung if the number of CRC errors on a router interface exceeds one error per second. Also shown on this monitor are vital statistics about the interface such as total packets input and output, total broadcasts received, and total bytes input and output.

The "Monitor Gateway" function has helped to solve many of the more subtle network problems. Many times the route between two routers appeared to be functioning correctly. However, when the router is more closely examined with this function, usually at least one interface is found to have an exceedingly high error rate, or some other non-obvious error. A network engineer can see the error rate on the graph and determine if there is a problem with a specific interface.

The second area involves the "Monitor Network" function. The "Monitor Network" function can determine the current connectivity from the source location to the rest of the HP NET Internet. This function can also provide a more detailed examination of the router's performance and status.

Connectivity is tested by simply performing a "ping" to all the routers at the various sites. If they all respond, the label for the site is colored green. If any router should fail to respond, the label is colored red. If the IP address of the router cannot

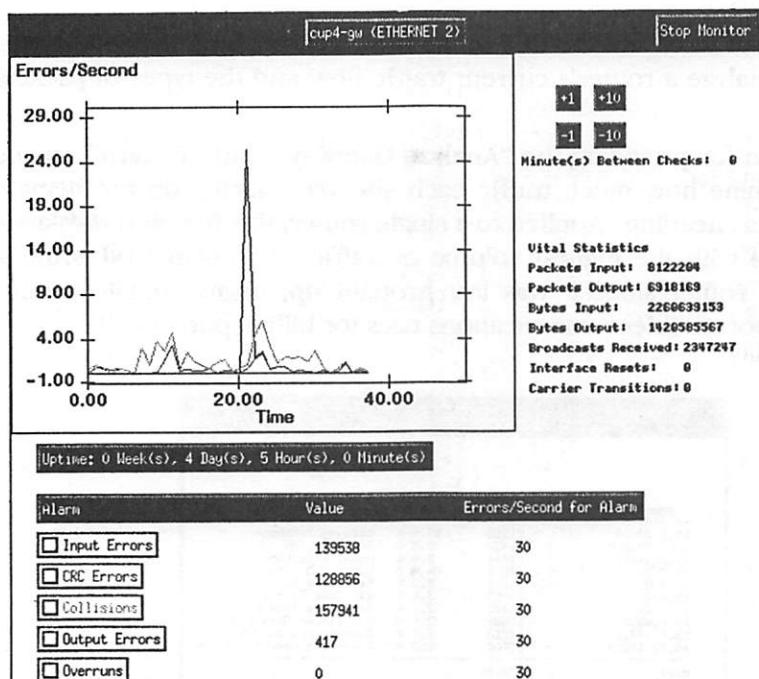


Figure 4: Xpath's Monitor Gateway display

be determined, the label is colored yellow. This functionality thus provides a rapid method for discovering unreachable routers on the network.

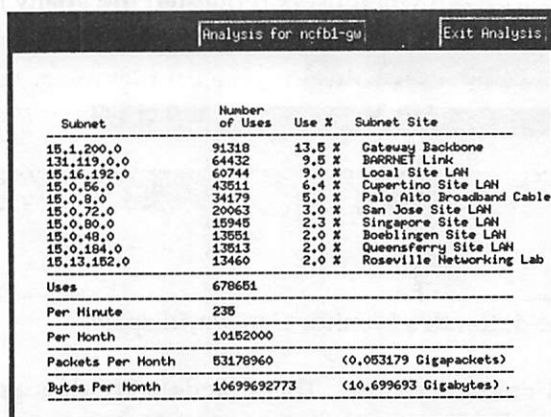
More detailed monitoring is typically only done on those routers administered by a specific group or site. When instructed, xpath will check the status and error rates on all the interfaces of the selected routers. If the interface is down, the label will be colored magenta. If the interface's error rate exceeds a user-specified threshold, the label is colored light blue. By selecting the site label, a window displaying the routers is displayed. The status of each router corresponds to the background color of the icon. Further information about a specific router, such as the status and error rates of its interfaces, can be found by selecting the button containing the name of the router. If the router icon itself is selected, the network engineer is provided with a session window to the router, allowing the engineer to issue configuration commands or to query the router for specific information.

The "Monitor Network" functionality is perhaps the most widely used portion of xpath. At the HP NET Internet Network Control Center, xpath runs the "Monitor Network" function twenty-four hours a day, keeping track of the error rates of the local routers as well as the connectivity from the Control Center to all other HP NET Internet sites. When a problem occurs, xpath immediately alerts the network operators. In many instances the problem can be fixed even before user complaints are registered on the Internet Hotline. Because of this, the process of network troubleshooting has gone from a problem-driven method to a proactive method.

#### 4. Analysis Functions

It is often useful to be able to determine how the routers are being used. Xpath has the ability to analyze a router's current traffic flow and the types of packets being routed.

The motivation for providing the "Analyze Gateway Traffic Pattern" function was the need to determine how much traffic each site was placing on the network, and where this traffic was heading. Applied to a single router, this function displays the ten destination subnets with the highest volume of traffic. Also displayed is the average traffic through the router since it was last brought up, normalized to a one-month period, which Corporate Telecommunications uses for billing purposes.



Subnet	Number of Uses	Use %	Subnet Site
15.1.200.0	91318	13.5 %	Gateway Backbone
131.119.0.0	64432	9.5 %	BARRNET Link
15.16.192.0	60744	9.0 %	Local Site LAN
15.0.56.0	43511	6.4 %	Quertino Site LAN
15.0.8.0	34179	5.0 %	Palo Alto Broadband Cable
15.0.72.0	20063	3.0 %	San Jose Site LAN
15.0.80.0	15945	2.3 %	Singapore Site LAN
15.0.48.0	13551	2.0 %	Boeblingen Site LAN
15.0.184.0	13513	2.0 %	Queensferry Site LAN
15.13.152.0	13460	2.0 %	Roseville Networking Lab
<b>Uses</b>	<b>678651</b>		
<b>Per Minute</b>	<b>235</b>		
<b>Per Month</b>	<b>10152000</b>		
<b>Packets Per Month</b>	<b>53178960</b>	(0.053179 Gigapackets)	
<b>Bytes Per Month</b>	<b>10699692773</b>	(10.699693 Gigabytes)	

Figure 5: Xpath's Analyze Gateway Traffic display

A network engineer may also be interested in the type of packets passing through the router. By using the "Analyze Gateway Traffic Type" function, the network engineer can get statistics on packet types such as IP broadcasts, ICMP redirects, and, on the HP NET Internet, Probe<sup>4</sup> packets. The information gathered from this function has helped solve many networking problems including finding UDP checksum errors, false ICMP redirects, and incorrect IP fragmentation.

Corporate Telecommunications has the responsibility to manage the links that connect the major sites on the HP NET Internet. Since user performance is directly related to the current utilization of these links, these links must be monitored and analyzed to determine if there is adequate network bandwidth.

The "Analyze Backbone Load" function produces an abstract map showing the major HP NET Internet sites and the backbone links connecting them. The thickness of the lines in the window correspond to the bandwidth of the actual communication lines. Each backbone link is color-coded based on the current utilization percentage.

Corporate Telecommunications is using this function to determine if the HP NET Internet backbone links are being used efficiently. This function is also being used in planning for bandwidth acquisition and in determining future network topology.

<sup>4</sup> Probe is an HP proprietary method of translating host names to IP addresses, and IP addresses to 802.3 addresses.



By using the "Analyze Backbone Load" function, Corporate Telecommunications' network engineers have been able to manage the HP NET Internet backbone, balancing the bandwidth utilization to match the capacity. In one instance, one subnet was linked to the rest of the network by a 512 Kilobits/sec circuit and a T-1 (1.544 Megabits/sec) circuit. It was discovered that all traffic destined for this subnet was routed through the T-1 circuit, which had a 73% utilization. In contrast, the other circuit's utilization was only 2%. After changing the routing metrics on the T-1 circuit, the utilization percentages more evenly matched each circuit's capacity.

## 5. LAN Functions

These functions allow xpath to be used at the LAN level instead of the WAN level. This functionality was added to xpath when users requested the ability to look at their local devices as well as observing the network as a whole.

The "LAN Builder" allows the user to draw a logical picture of the LAN. Allowable nodes include HP 9000 systems, HP 3000 systems, and cisco Systems routers.

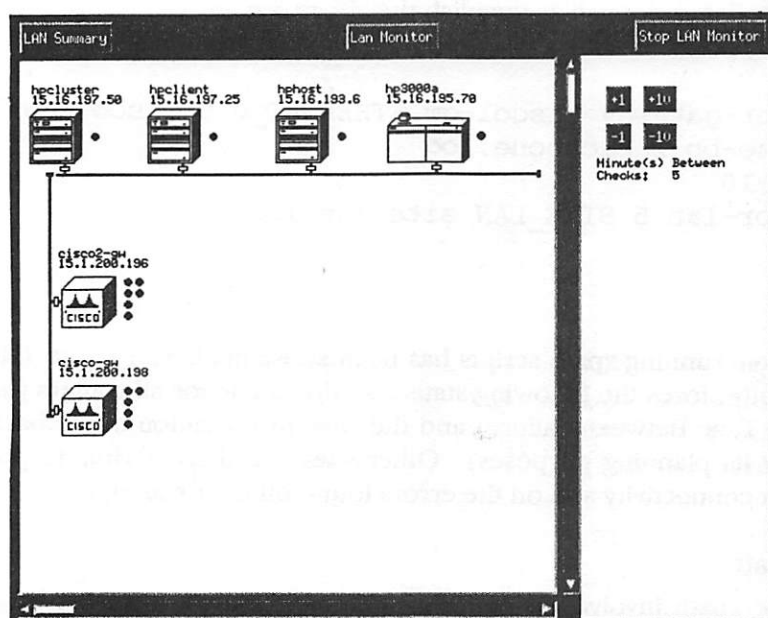


Figure 6: Xpath's LAN Monitor display

Once built, the "LAN Monitor" function operates similarly to the "Monitor Network" function, but uses the picture built by the user instead. Xpath imparts network connectivity information by drawing a series of circles next to each cisco Systems router for each valid interface on the router. For each host on the LAN one circle is drawn next to it. Each cisco Systems router's interface is checked and color-coded according to its current status: green means the interface is up, yellow means it is administratively down, and red means it is down. Likewise, for each host, a "ping" is sent to the host and the circle is changed to display its current status: green means the host responded, red means it did not, and yellow means that the host's IP address cannot be determined.



While the "LAN Functions" do not incorporate new algorithms into xpath, they do provide a way to obtain and record information about critical systems on a LAN. This functionality has been used extensively by HP NET Internet sites to obtain information about router interface conditions, as well as the status of local critical systems such as diskless servers, mail hubs, etc. Although this functionality is similar to xnetmon, though there are some differences, mostly in the information provided about the hosts and routers.

## 6. Script Functions

This area was developed to automate portions of xpath. A special language was developed to control the execution of xpath. Many of xpath's functions can be run from an xpath script with the output going to an ASCII file. One example might be to repeat the following sequence 10 times: monitor the router *cisco1-gw*'s Ethernet 0 interface for five iterations, analyze the network backbone for two iterations, sleep for 10 minutes, monitor the LAN *SITE\_LAN* for five iterations, then finally terminate. The output from these commands will be placed in various files that can be examined at a later date. The following script will accomplish the above sequence:

```
loop 10
    monitor-gateway cisco1-gw ETHERNET_0 5 cisco1-gw.log
    analyze-bb 2 backbone.log
    sleep 10
    monitor-lan 5 SITE_LAN site_lan.log
loop-end
exit
```

The results from running xpath scripts has been substantial. Corporate Telecommunications currently stores the following statistics: downtime for all routers (used to calculate the Mean Time Between Failure) and the current utilization of the backbone links is stored (for its planning purposes). Other sites use these scripts to produce reports on network connectivity and on the errors found on their routers.

## Future work on Xpath

The future for xpath involves using SNMP (Simple Network Management Protocol) to access the network elements and add functionality to the current program.

Currently there is an SNMP interface that xpath can use. However, less than 15% of the routers on the HP NET Internet understand SNMP. Once a larger percentage of the cisco routers have their system software updated, xpath will rely on SNMP to query the routers.

Another area for future xpath development will deal with router configuration without needing to understand the router's command set. This would eliminate the need for the network engineer to know the command set of every router on the network.

## Conclusions

One may wonder, given the development of network management protocols, why we wrote our programs. The reason is simple. At the time, there were no real solutions to the problem of managing a large TCP/IP network. These programs were written to solve this problem, both at the LAN and WAN level. Currently, both *xnetmon* and *xpath* are widely used by network administrators within HP.

*Xnetmon* should easily port to almost any UNIX system that allows ICMP packets to be generated, and will monitor any network client that correctly responds to the ICMP "ping". On the other hand, it might be difficult to port the current version of *xpath*. This is not because of any programming difficulties, but because of the way *xpath* gathers its data. Since there is no programmatic interface to a cisco router, all statistics from them are acquired by logging onto the gateway and executing commands to acquire the desired information. Unless other vendors' gateways provide similar information, or they understand SNMP, *xpath* might not easily fit into a more heterogeneous network.

The recent proliferation of network management platforms and devices will most likely shortly spell the demise of both *xnetmon* and *xpath*. However, until these are widely available and provide the necessary functionality, these two programs will play a major role in the management of the HP NET Internet.



Allan S. Leinwand  
Hewlett-Packard



Jeff Okamoto  
Hewlett-Packard

Allan S. Leinwand received the B.S. degree in Computer Science from the University of Colorado at Boulder in 1988. Currently, he is working on a M.S. degree in Computer Science part-time from Stanford University. While an undergraduate, he worked with CAD/CAM software at Auto-trol Technology in Thornton, Colorado and developed a user interface for a software project with the Martin Marietta Aerospace in Denver, Colorado. Since graduation, Allan has been a member of the Hewlett-Packard Corporate Telecommunications Department working on the HP NET Internet. He is involved in network planning, network support and the numerous tasks that are involved in running a large TCP/IP network. Also, Allan has been involved in writing documents and procedures for the internal Hewlett-Packard networking security standard and has helped influence the product divisions toward the development of a network management product.

Jeff Okamoto received his B.A. degree in Computer Science from the University of California at Berkeley in 1986. Since graduation he has been a member of the Corporate Computing and Services department of Hewlett-Packard. His job duties include porting engineering application, maintaining the flow of electronic mail and netnews, and administering the group's diskless cluster.

He is spending much of his time analyzing how CC&S's high-end computing resource will fit into HP's vision of a Cooperative Computing Environment, in which CPU-intensive engineering application can be run on CC&S's computing engines, which are accessed via HP's TCP/IP network.



# Traffic Characterization of the NSFNET National Backbone \*†‡

Steven A. Heimlich

Department of Computer Science  
University of Maryland  
College Park, MD 20742

September 1989

## Abstract

Traditionally, models of packet arrival in communication networks have assumed Poisson arrival patterns. An earlier study of a local area network at MIT found that traffic followed a more general model, called the "packet train."

This study verifies the existence of packet trains on the NSFNET national backbone network. Packet train characteristics are categorized by application and transport protocols, network topology (traffic which is local to the monitored node's regional network or in transit), and level of utilization. The strength of the model is shown to be related both to utilization and to protocol.

Performance of protocol implementations may be improved by taking advantage of packet trains, performing per train rather than per packet processing. In addition, route prefetching based on the existence of packet trains is shown to be more effective than caching destination addresses alone.

## 1 Introduction

Traditionally, models of packet arrival in communication networks have assumed either Poisson or compound Poisson arrival patterns. A study of a token ring local area network (LAN) at MIT [7] found that packet arrival followed neither of these models. Instead, traffic followed a more general model dubbed the "packet train," which describes network traffic as a collection of packet streams traveling between pairs of nodes. A packet train consists of a number of packets travelling between a particular node pair.

Mean train interarrival time was found to be much larger than mean interarrival time for packets within a single train. Furthermore, successive packets on the MIT network tended to belong to the same trains, a phenomenon called "locality." Measured locality varied inversely with respect to network utilization.

A more recent study of the Carnegie Mellon campus backbone and computer science department local network [8] confirmed that a small number of node pairs generate a large

---

\*Copyright © 1989 by Steven A. Heimlich.

†Work supported by the IBM Corporation under Agreement #15050042.

‡Author's current address: heimlich@ibm.com

proportion of network traffic, thus reaffirming the existence of locality on local networks. Gusella's study of a Berkeley departmental Ethernet [5] and preliminary investigations at the University of Maryland of the computer science department Ethernet confirmed that packet interarrival times on local networks do not follow Poisson arrival patterns.

This study verifies the existence of packet trains on NSFNET, a high speed, centrally managed national backbone network. While the train characteristics are not as striking as those found on the MIT local network, they are quite strong given the great number of hosts communicating through the backbone.

## 1.1 Motivation

As the speed of physical communication links, the number of interconnected networks, and the number of distributed applications all increase, the need for ever faster (perhaps more sophisticated) packet switching software grows. Knowledge of the existence of packet trains could lead to reduced delay in packet switches.

The notion of train locality (defined below), if confirmed on NSFNET, may lead to a route caching strategy more successful than the current procedure of consulting a complete route table for every packet, particularly when implemented using associative memory.

In addition, it may be possible to provide better queue management within an IP implementation than the single packet queue currently in use [18]. There are various proposals for alternate IP queueing strategies, ranging from three queues — login, FTP, and other traffic — to one queue for each active packet source [11]. A strategy based on the existence of packet trains may be a reasonable compromise. Such a strategy could provide a more effective approach to congestion avoidance and a fairer policy of congestion control than currently implemented [18].

The existence of packet trains affects protocol interface design as well. Song and Landweber [19] discuss the benefits of performing per train rather than per packet processing in network adapter interface design.

The following section describes the NSFNET backbone and the measurement environment. The remaining sections describe the packet train model, monitoring and analysis software, results of backbone monitoring, and conclusions.

## 2 Environment

### 2.1 Hardware

NSFNET is a DS-1<sup>1</sup> speed national backbone Internet Protocol (IP) [15] internetwork connecting several regional computer networks and supercomputer centers. Currently, the backbone consists of 13 production nodes and 4 test nodes. Nodes (called Nodal Switching Systems—NSS) communicate via both terrestrial fiber and microwave links, and each production NSS has between three and five logical links into the backbone and one link to the connected regional.

Each NSS (Figure 1), in turn, consists of several IBM RT workstations. Intra-NSS communication takes place over a 4 megabit/sec token ring LAN. The RTs perform two basic functions: packet switching and dynamic routing. Among the packet switches are several internal RTs which direct traffic through the backbone, and an external RT which

---

<sup>1</sup>1.544Mbps



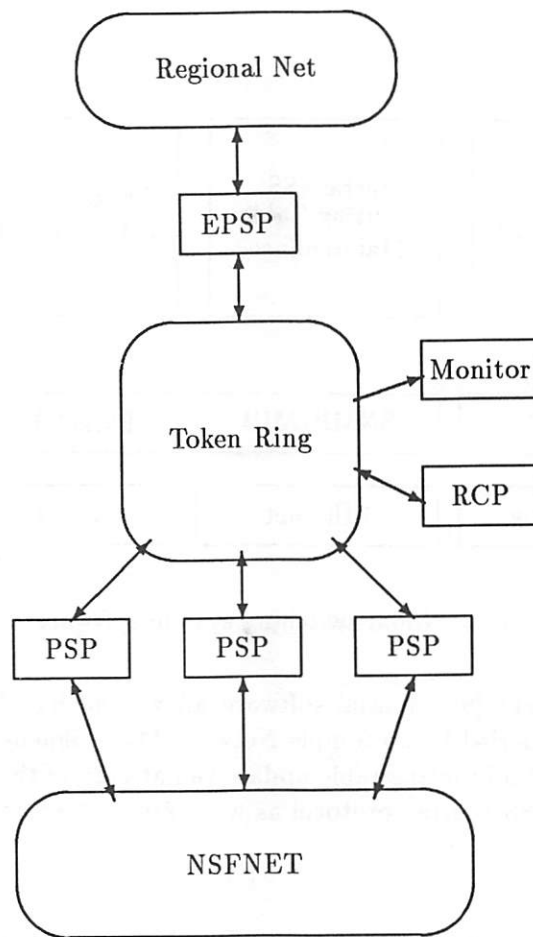


Figure 1: A typical nodal switching system.

delivers traffic to and receives traffic from the connected regional network. The external packet switch communicates to the regional network via an Ethernet. An additional RT, the routing control processor (RCP), at each NSS runs the routing protocol and forwards routing tables to each of the NSS's packet switches. Packets flowing through the backbone enter via the external packet switch (EPSP), traverse several interior packet switches (PSP), and exit via another EPSP.

The Ann Arbor, Michigan, NSS contains another RT which contains promiscuous token ring hardware used to monitor traffic on that node. The RT has a 1KHz clock giving one millisecond resolution, easily adequate for monitoring packet trains on NSFNET. Given the loosely coupled architecture and current packet processing delay [18], it is not possible for consecutive packets from the same train to arrive within one millisecond of each other.

## 2.2 Packet Switch Software

All of the RTs in the backbone run a special version of the 4.3BSD Unix operating system. The kernel contains drivers for IBM ARCTIC cards (to drive the point-to-point links), token ring (for intra-NSS traffic) and Ethernet (for regional traffic). In addition, there are extensions to the drivers and the IP layer to support the Management Information Base

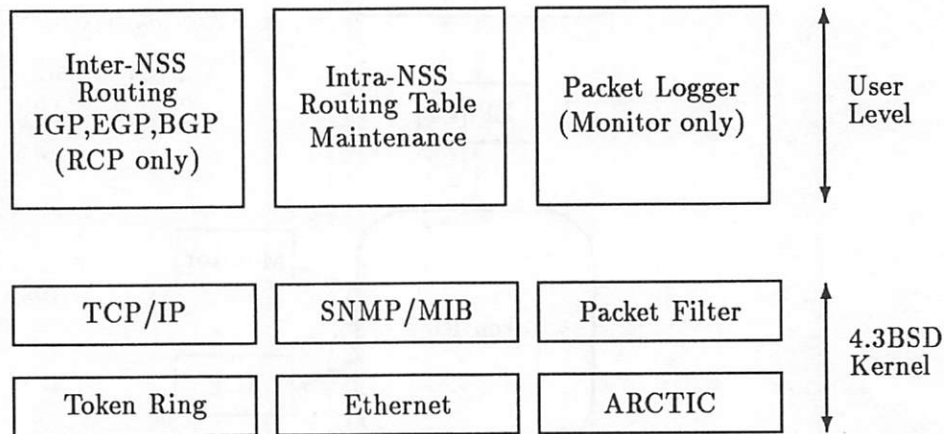


Figure 2: Nodal switching system software.

(MIB) for TCP/IP networks [9]. Special software allows the RTs in a given NSS to be seen as one entity when queried by an Simple Network Management Protocol (SNMP) [2] client. Daemons for intra-NSS routing table update run at each of the RTs, and the routing processor runs the inter-NSS routing protocol as well. Figure 2 shows NSS software.

## 2.3 Protocols

While NSFNET carries a great variety of IP traffic, this study concentrates on the most common (in terms of number of packets counted at the Ann Arbor NSS) transport and application protocols. In particular, train characteristics of the Transmission Control Protocol (TCP) [16] and User Datagram Protocol (UDP) [12] transport protocols are presented. Additionally, the Simple Mail Transfer Protocol (SMTP) [17], domain nameserver, File Transfer Protocol (FTP) [14], SNMP and Simple Gateway Monitoring Protocol (SGMP) [3], Telnet [13] and rlogin (terminal traffic) application protocols are classified.

TCP is a connection-oriented sliding window transport protocol which guarantees in-order delivery of a stream of packets from a user. UDP is a datagram transport protocol which has does not retransmission and makes no guarantee of packet delivery. Telnet, rlogin, FTP, and SMTP all use TCP; SGMP, SNMP, and the domain nameserver queries all use UDP. SGMP and SNMP, and domain nameserver queries are straight forward request-response type protocols, while SMTP does its own application level connection setup and teardown in addition to delivering mail data.

## 2.4 Difficulties

### 2.4.1 Packet Loss

Although traffic on the backbone has been growing steadily since its inception, switching capacity easily exceeds observed load. Thus while utilization is quite high (up to 1,200,000 packets/hour observed on the Ann Arbor NSS — see Figure 3), this study considers behavior

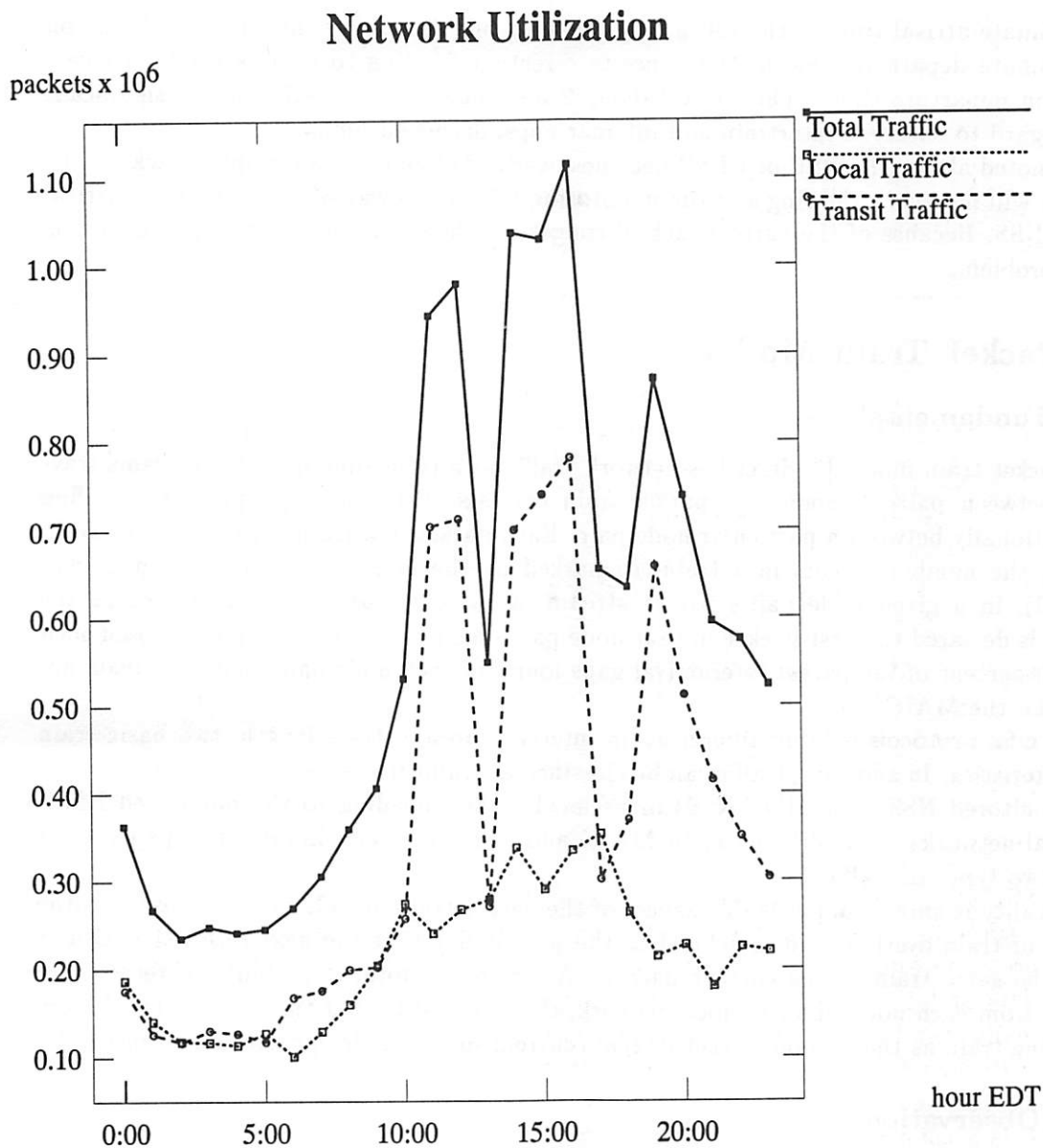


Figure 3: Utilization over a 24-hour period.

measured on an uncongested backbone.

Possible sources of packet loss are the token ring hardware and the limited data space at the user/kernel level boundary. During this study the token ring driver on the monitoring RT, which is the most highly utilized of all the RTs on the Ann Arbor NSS because it operates in promiscuous mode, has reported zero packet loss. In addition, the packet filter has not dropped a single packet due to lack of buffer space in the kernel.

#### 2.4.2 NSS Architecture

Because of the loosely coupled NSS architecture, the monitoring RT can listen to only the token ring for packets traversing the NSS. Therefore, the packet timestamps are based on

approximate arrival time at the token ring driver in one direction while they are based on approximate departure time in the opposite direction, leading to some skew for packets based on departure time. This skew (about 2 ms), however, is statistically insignificant with regard to observed intertrain and intercar gaps, discussed below.

As noted above, if a PSP or EPSP becomes overloaded and starts dropping packets, the packets will not reach the ring and the monitoring RT has no way of recording their arrival at the NSS. Because of the current lack of congestion, however, this type of packet loss is not a problem.

### 3 Packet Train Model

#### 3.1 Fundamentals

The packet train model [7] describes network traffic as a collection of packet streams traveling between pairs of nodes. A packet train consists of a number of packets traveling bidirectionally between a particular node pair. Each packet in a train is called a car. Train length, the number of cars in a train, is marked by the maximum allowed intercar gap (MAIG). In a given node pair's packet stream, a packet whose arrival gap exceeds the MAIG is declared the first packet in that node pair's next train. The MAIG is chosen such that 90 percent of the packet interarrival gaps found in each node pair's packet stream are less than the MAIG.

Specific protocols exhibit differences in intercar gap and train length, two basic train characteristics. In addition, traffic can be classified as traffic in transit (just passing through the monitored NSS to another NSS) or as local traffic (heading to the monitored NSS's regional network); clear differences in MAIG and in intercar and intertrain gaps exist for these two types of traffic.

Locality is another quantifiable aspect of the packet train model. Locality describes the extent of train overlap, and is defined as the probability that the next received packet is from the same train as the current packet. Assuming uniform probability of receiving a packet from each node in an  $m$  node network, the probability that the next packet is from the same train as the current packet is  $2/m$  (current direction or opposite direction).

#### 3.2 Observations

In fact, Jain observed that this probability was 60% on the MIT LAN (a network with 33 hosts). Preliminary studies of the Maryland Computer Science Department LAN show that during low utilization locality for traffic through the LAN's router is about 60%, while during high utilization it is 40%. For traffic which does not flow through the router, locality during low utilization is about 35%, while during high utilization it is 28%. The lower figures found on the Maryland LAN can be attributed to the high utilization. At MIT in 1985, Jain measured 11 million packets over a period of 1 week. The LAN at Maryland carries about 300,000 packets in a single hour during low utilization and approximately 800,000 packets in an hour during high utilization. The environment at Maryland contributes to the astonishingly high locality; there are over 120 workstations, most of which use the LAN to reference a small number of file servers. Accordingly, cache hit ratios obtained in this environment (see Tables 1 and 2) are quite high. Because of the high utilization and the great number of communicating hosts using the backbone, locality is much less. However,

Router Traffic				
cache size	Low Util		High Util	
	prefetch	dest	prefetch	dest
2	0.64	0.47	0.48	0.30
8	0.92	0.86	0.84	0.75
16	0.96	0.93	0.93	0.88
24	0.97	0.96	0.96	0.93
32	0.98	0.97	0.97	0.95
40	0.99	0.98	0.98	0.96

Table 1: Cache hit ratios for traffic flowing through the router on the UMD LAN.

Non-router Traffic				
cache size	Low Util		High Util	
	prefetch	dest	prefetch	dest
2	0.50	0.34	0.38	0.27
8	0.72	0.65	0.73	0.64
16	0.93	0.84	0.92	0.84
24	0.96	0.92	0.96	0.91
32	0.98	0.95	0.98	0.95
40	0.99	0.97	0.99	0.97

Table 2: Cache hit ratios for traffic local to the UMD LAN.

locality still varies inversely with respect to network utilization. Furthermore, traffic in transit exhibits less locality than local traffic because of the probable greater number of communicating hosts and the high degree of merged traffic.

## 4 Special Software

### 4.1 Monitoring Software

The monitor RT at the Ann Arbor NSS runs the Stanford Packet Filter software [10] just above the token ring driver. This software runs as a pseudo-device and buffers all packet headers traversing the Ann Arbor ring. Headers are timestamped in the kernel as they are buffered. The NNStat package [1] can run above this packet filter (outside the kernel) to provide traffic protocol statistics. In order to monitor the traffic for this study, however, special software runs at the user level and processes headers logged by the packet filter, writing relevant information to disk.

This software treats the packet filter as a producer, blocking until the filter has packets to deliver, then processing individual packet headers until the packet filter buffers are empty. Non-IP packets are counted but not processed; IP packets with the wrong version number are ignored. Packets headed to or from the token ring address of the EPSP are marked local; all other packets are deemed transit.



## 4.2 Analysis Software

Analysis software runs offline. This software reads a dump produced by the monitoring software and presents either an interarrival time histogram based only on arrival time (giving the Poisson arrival statistics) or on packet trains. The packet train analysis runs in two passes. Pass one scans the dump and builds an interarrival time histogram based on arrival time and train (source and destination addresses concatenated). Pass two calculates the MAIG such that 90% of each packet stream's interpacket gaps are less than this value and scans the dump again, calculating mean train length. The MAIG calculated for all traffic is supplied to the program when generating statistics for particular protocols.

## 5 Results

Traffic log analysis for both the Poisson arrival and packet train models is categorized by application protocol, transport protocol, utilization, and topology. Utilization for this analysis is defined as the number of packets counted disregarding protocol. As shown in Figure 3, high utilization for transit traffic occurred during hours 11, 12, 14, 15, and 16; low utilization occurred during hours 2, 3, 4, 5, and 6. For local traffic, high utilization occurred during hours 14, 16, and 17; low utilization occurred during hours 2, 3, 4, 5, and 6.

### 5.1 Fit to Poisson Arrival Model

In addition to analyzing the fit for all protocols, traffic is broken down by application protocol (FTP, domain nameserver and telnet and rlogin) and transport protocol (TCP and UDP). The samples of SNMP/SGMP are too small to warrant analysis. All figures present  $\log(\text{number of packets})$  for interarrival gaps of 0 to 500 ms. Interarrival gap for all traffic can also be considered to be the idle time of the appropriate PSPs; the low likelihood of long periods of idle time is reflected in the increasing noise in the graphs. Gaps of more than 500 ms are not included in the graphs. Mean interarrival gaps and coefficients of variation are presented in Tables 3 and 4.

Figure 4 presents the interarrival time distribution for all traffic. As shown, utilization is the dominating factor in the slope of the curves; whether the traffic is local or in transit is insignificant. During periods of both high and low utilization, the shape of the curve (when plotted on a log scale) is basically a straight line, an indication of an exponential distribution. The slope of the curve is greater for high than for low utilization, indicating less idle time. Particularly for periods of high utilization, coefficients of variation are close to 1.0, the coefficient for an exponential distribution.

Figure 5 shows domain nameserver traffic. With the exception of transit traffic under low utilization, the curves are essentially straight lines. Coefficients of variation, however, are near 1.5 during periods of low utilization and higher for high utilization, possibly reflecting the variability in name server processing time under high load.

Ftp traffic, shown in Figure 6, clearly does not have an exponential distribution under low utilization, when statistics are more likely to be dominated by a single connection. For high utilization the fit is better, possibly because of the merging of traffic. Because the protocol tends to send large numbers of packets at once (unlike domain nameserver traffic, which is basically a transaction protocol) with a regular interpacket gap (because

Local Traffic				
protocol	High Util		Low Util	
	mean	cv	mean	cv
All	11	1.2	32	1.3
Domain	310	1.6	678	1.5
FTP	45	4.9	217	6.9
Login	36	1.4	314	2.9
TCP	12	1.2	32	1.3
UDP	136	1.7	243	1.7

Table 3: Mean interarrival times and coefficients of variation for local traffic.

Transit Traffic				
protocol	High Util		Low Util	
	mean	cv	mean	cv
All	5.3	1.3	28.2	1.3
Domain	36	2.1	132	1.5
FTP	23	3.4	207	6.1
Login	24	2.4	640	2.9
TCP	6.9	2.2	48	1.6
UDP	30	1.3	88	1.5

Table 4: Mean interarrival times and coefficients of variation for transit traffic.

most packets will be the same, maximum size), spikes occur at several locations, probably determined by distance (number of packet switches) of hosts from the backbone and regional network characteristics. Each regional network will smooth the interpacket gaps according to its own capacity, resulting in smaller gaps from hosts attached to higher capacity regionals and larger gaps from hosts attached to lower capacity regionals. The coefficient of variation for FTP traffic is rather high (about 7 during periods of low utilization).

The curve for login traffic (Figure 7) is nearly straight for high utilization. As with FTP traffic, the merging of traffic from different hosts may be responsible for this distribution. Under low utilization, however, the curve seems to fit two straight lines, with a spike around 200 ms. This spike could be an average command processing delay at destination hosts.

Like the curves for all traffic, the curves for TCP traffic (Figure 8) are nearly straight with a spike near 6 ms. The effect of utilization shows dramatically here in the slope of the curves. Under high utilization, the coefficient of variation for TCP traffic is very close to 1.0.

UDP traffic (Figure 9) as well shows a relatively straight curve despite the transaction processing nature of application protocols which might use UDP. Local traffic under low utilization particularly is quite variable, showing less of the affects of merging of traffic.

### 5.1.1 Why the Spike?

In looking at the interarrival distribution curves, the spike around 6 ms (especially during periods of low utilization) is a bit perplexing. During low utilization, when it is more likely

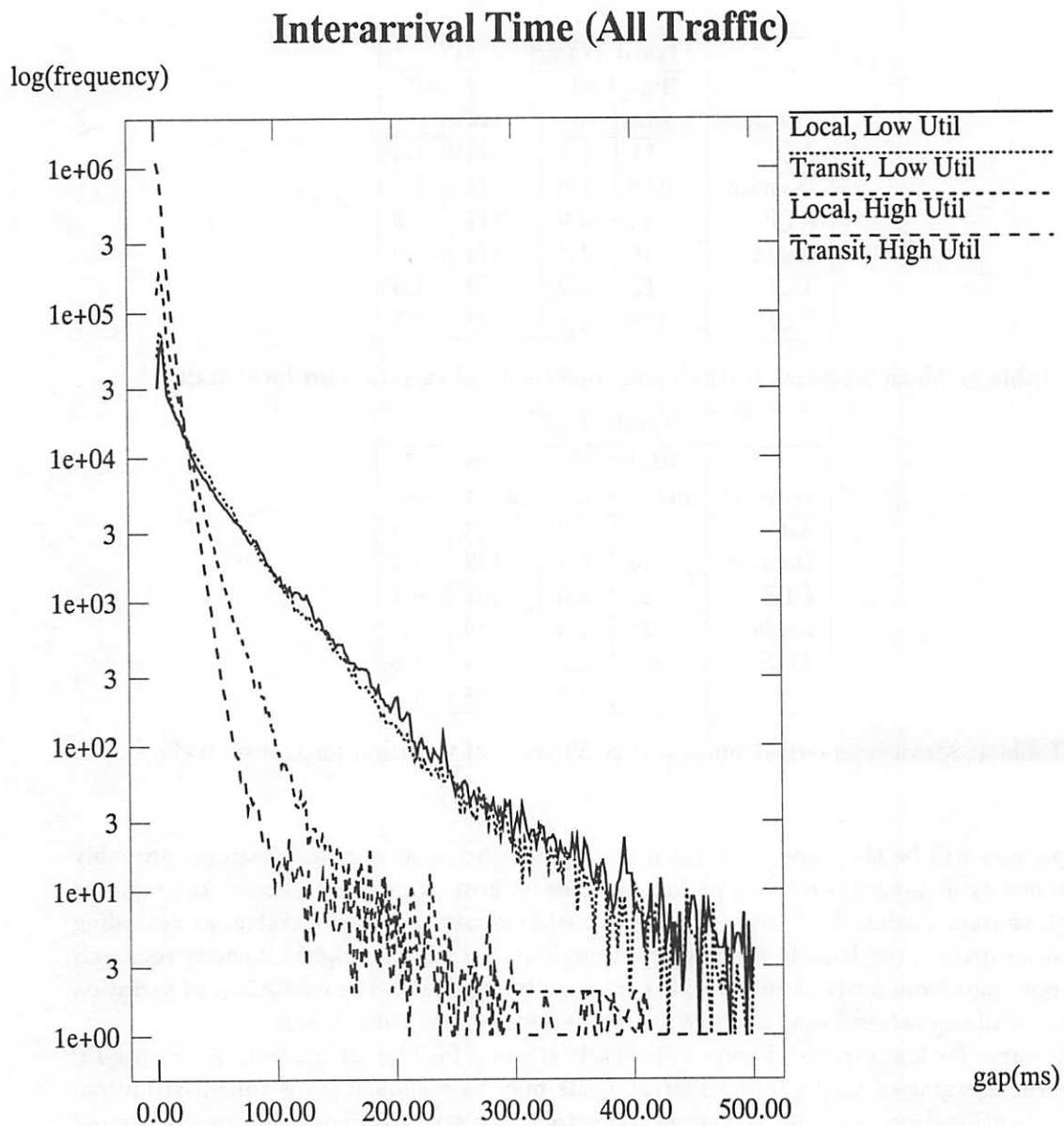


Figure 4: Interarrival time distribution for all traffic.

that the interarrival characteristics are dominated by only a few connections at once, it is possible that this spike is the result of draining packets from a single T1 line. Currently, there is a speed matching box at each end of every T1 line which introduces a delay of 3 ms per packet regardless of packet size [18]; these boxes (as well as packet switching delay at the RTs) skew the interarrival distribution toward 6 ms.

## Interarrival Time (Domain)

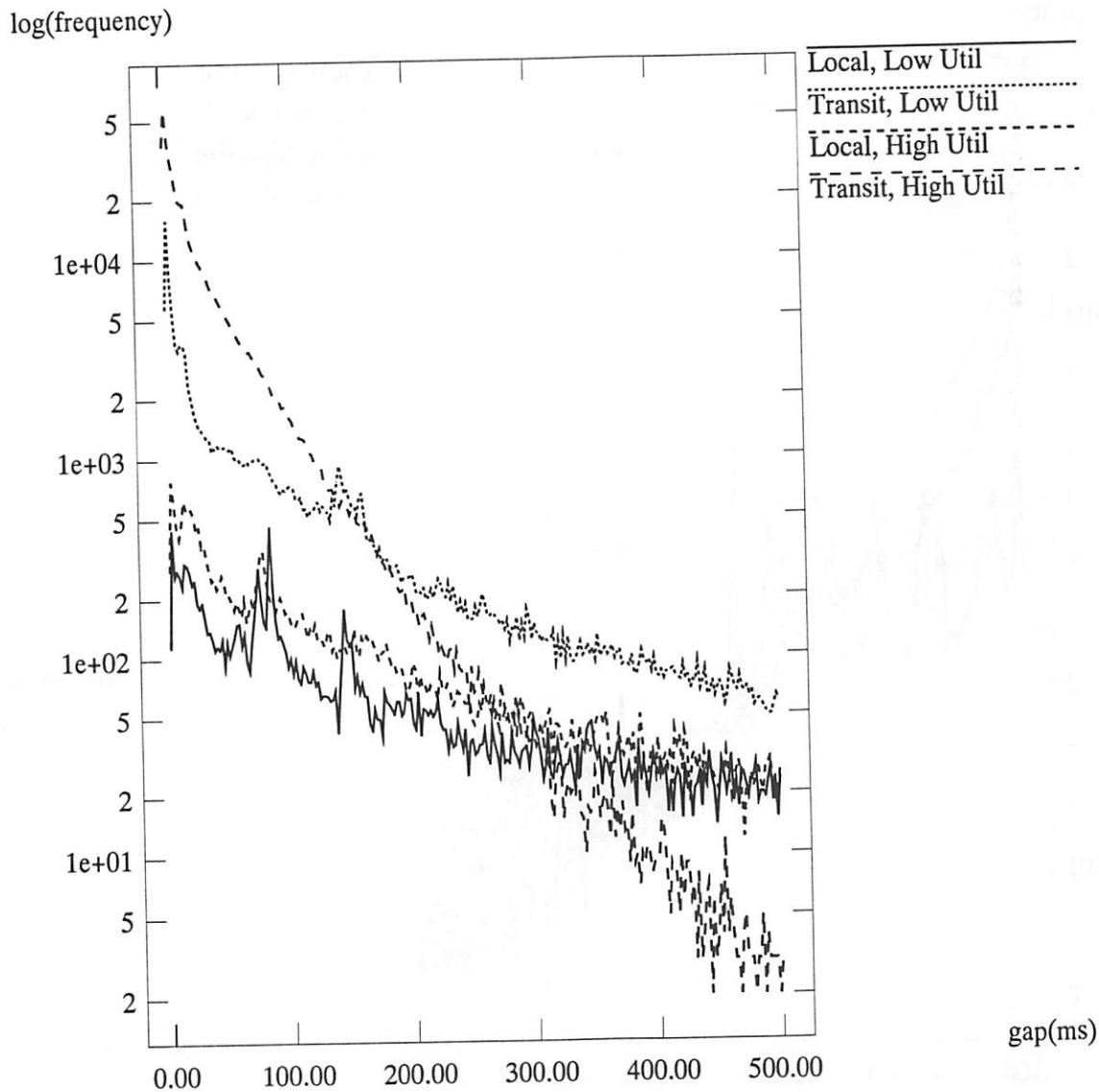


Figure 5: Interarrival time distribution for domain nameserver traffic.

## 5.2 Fit to Packet Train Model

### 5.2.1 Existence of Packet Trains

The existence of packet trains can be verified by measuring train locality on the backbone. Data shows, however, that while locality is reasonable during periods of low utilization — around 30% — it is rather low during periods of high utilization — around 10%. The low locality here does not necessarily rule out the existence of trains, however; it is more likely the result of the great number of hosts communicating through the backbone. In order to check for the existence of packet trains during periods of high utilization, the hit ratios of two different IP address caching schemes can be compared.

## Interarrival Time (FTP)

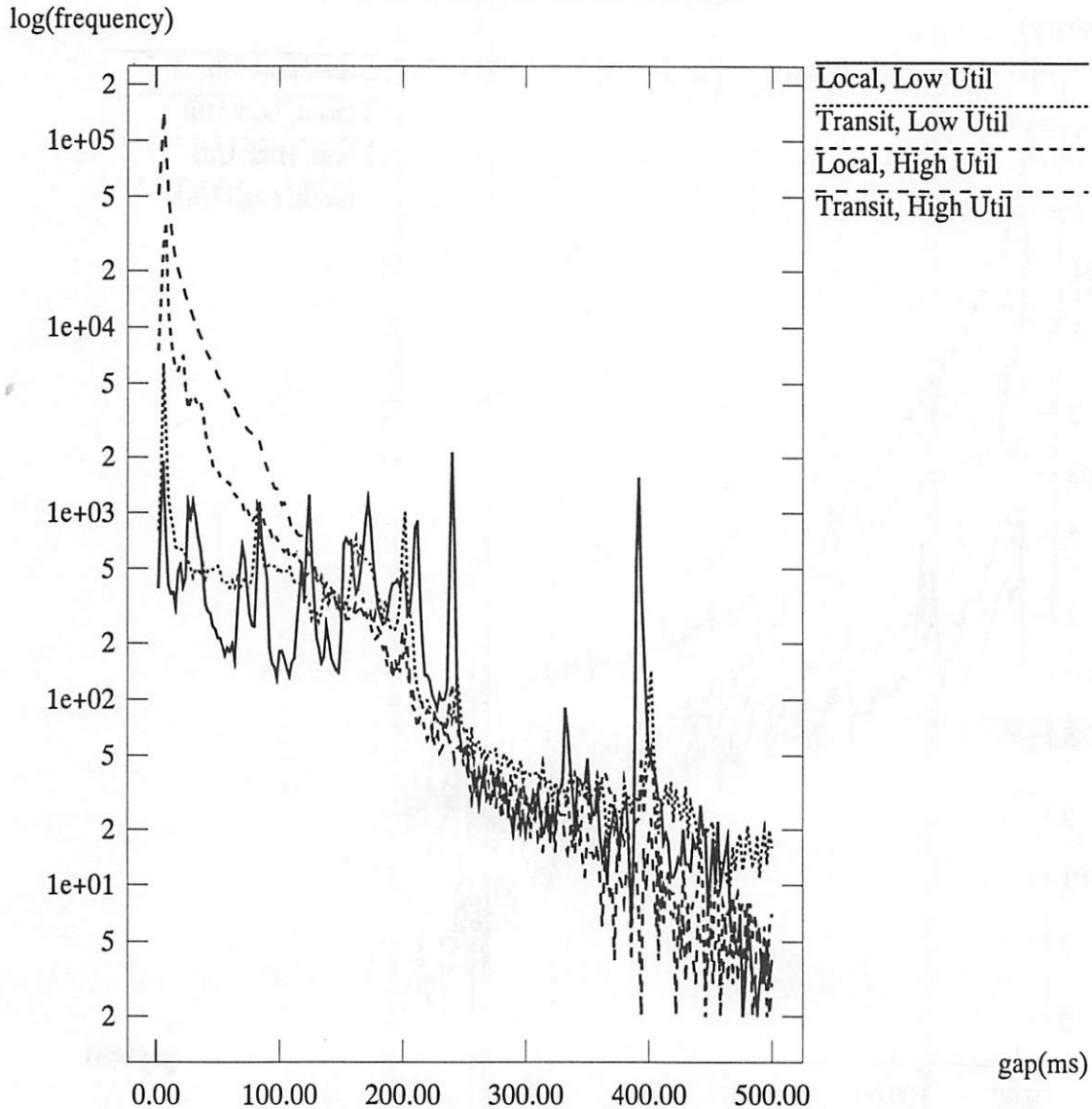


Figure 6: Interarrival time distribution for FTP traffic.

The first scheme, destination address caching, is based on destination addresses alone. Each cache size reflects the number of destinations stored in the cache (entries are deleted using an LRU algorithm). The alternate scheme, prefetching, caches both source and destination addresses; this strategy is based on the assumption that a packet from a given source will be followed by a packet destined to that source within a small amount of time. One half of the cache is used for sources and one half for destinations. For example, a cache size of eight allows for eight destinations using destination caching or four destinations and four sources using prefetching. As with destination address caching, entries in a prefetch cache are deleted using an LRU algorithm (except that the ratio of sources to destinations is kept 1:1). Packet trains exist if the prefetch hit ratios are greater than or equal to those for pure destination caching.



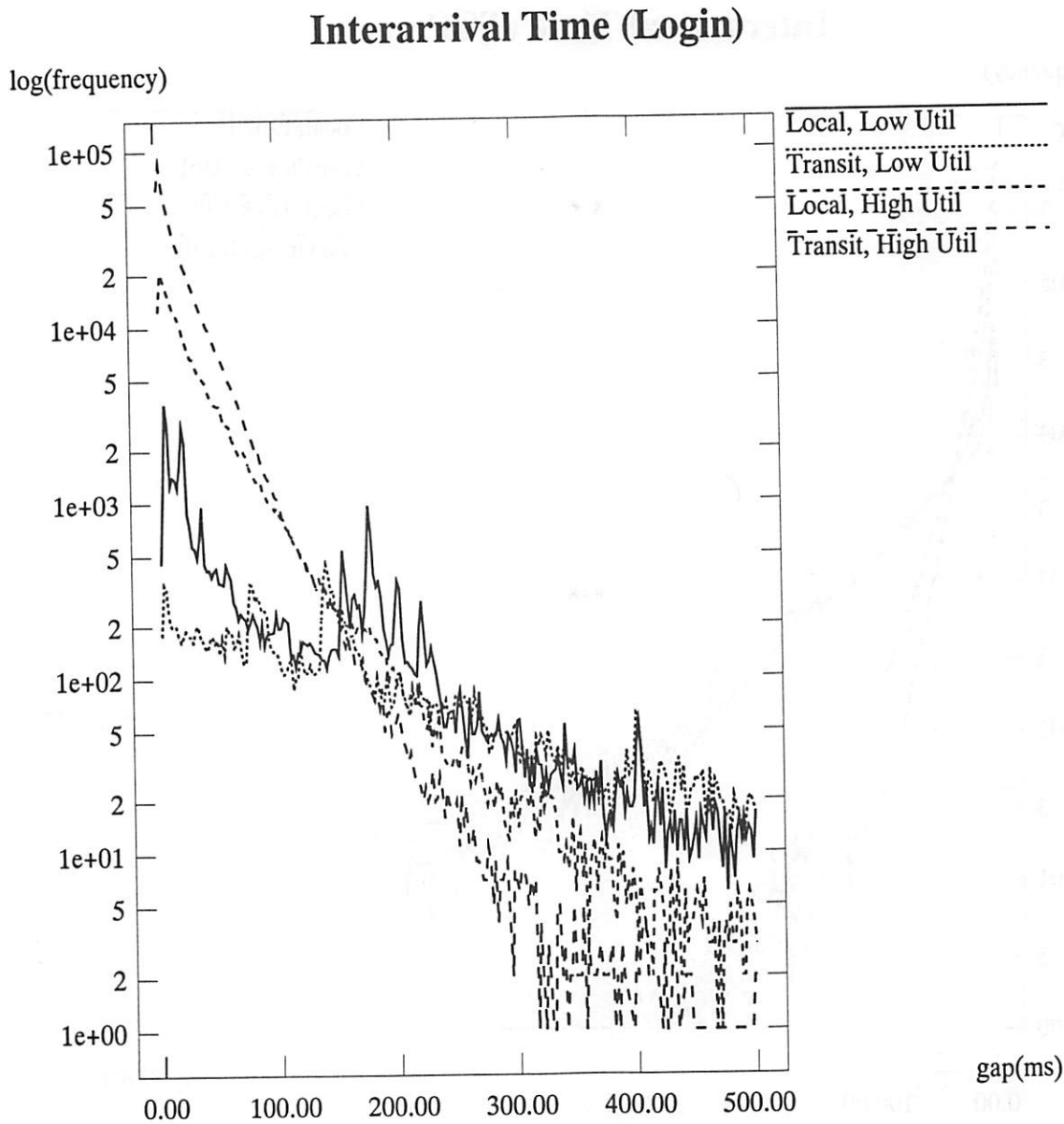


Figure 7: Interarrival time distribution for login traffic.

Tables 5 and 6 show hit ratios for both schemes for samples taken during periods of both high and low utilization. Prefetching source addresses yields hit ratios at least as high as caching destinations alone for all sizes. These figures are consistent over many different samples, and verify the existence of packet trains on the backbone. As revealed by these results, trains are less obvious for transit traffic because of the probable greater number of communicating hosts.

#### 5.2.2 Locality

Locality on the backbone is not as strong as on local networks. Jain [7] measured 60% locality on an MIT local network; investigations of the University of Maryland Department

## Interarrival Time (TCP)

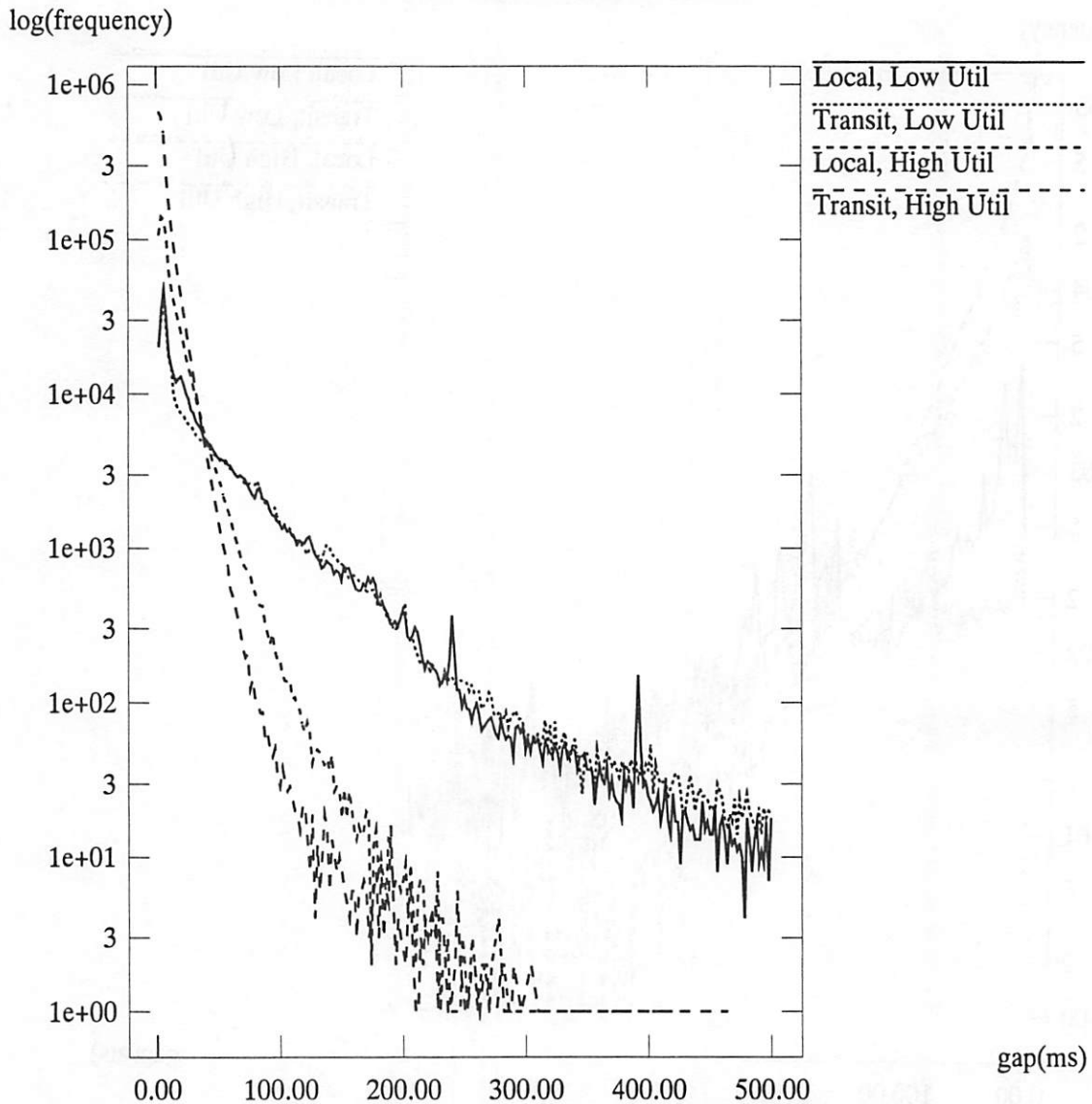


Figure 8: Interarrival time distribution for TCP traffic.

of Computer Science Ethernet prior to this study showed about 40% locality for traffic flowing into/out of the Ethernet and about 28% locality for traffic with both local source and destination. Locality on the backbone ranges from 8% to 37%. Given the possible number of hosts communicating at once, however, the very fact that locality can possibly reach 37% is quite extraordinary. Figure 10 shows the relationship of locality to utilization, clearly indicating their inverse relationship.

### 5.2.3 Train Parameters

**All Traffic** For all traffic, utilization affects mean intercar and intertrain gaps and maximum allowed intercar gap. Train length for all traffic throughout the day is relatively

## Interarrival Time (UDP)

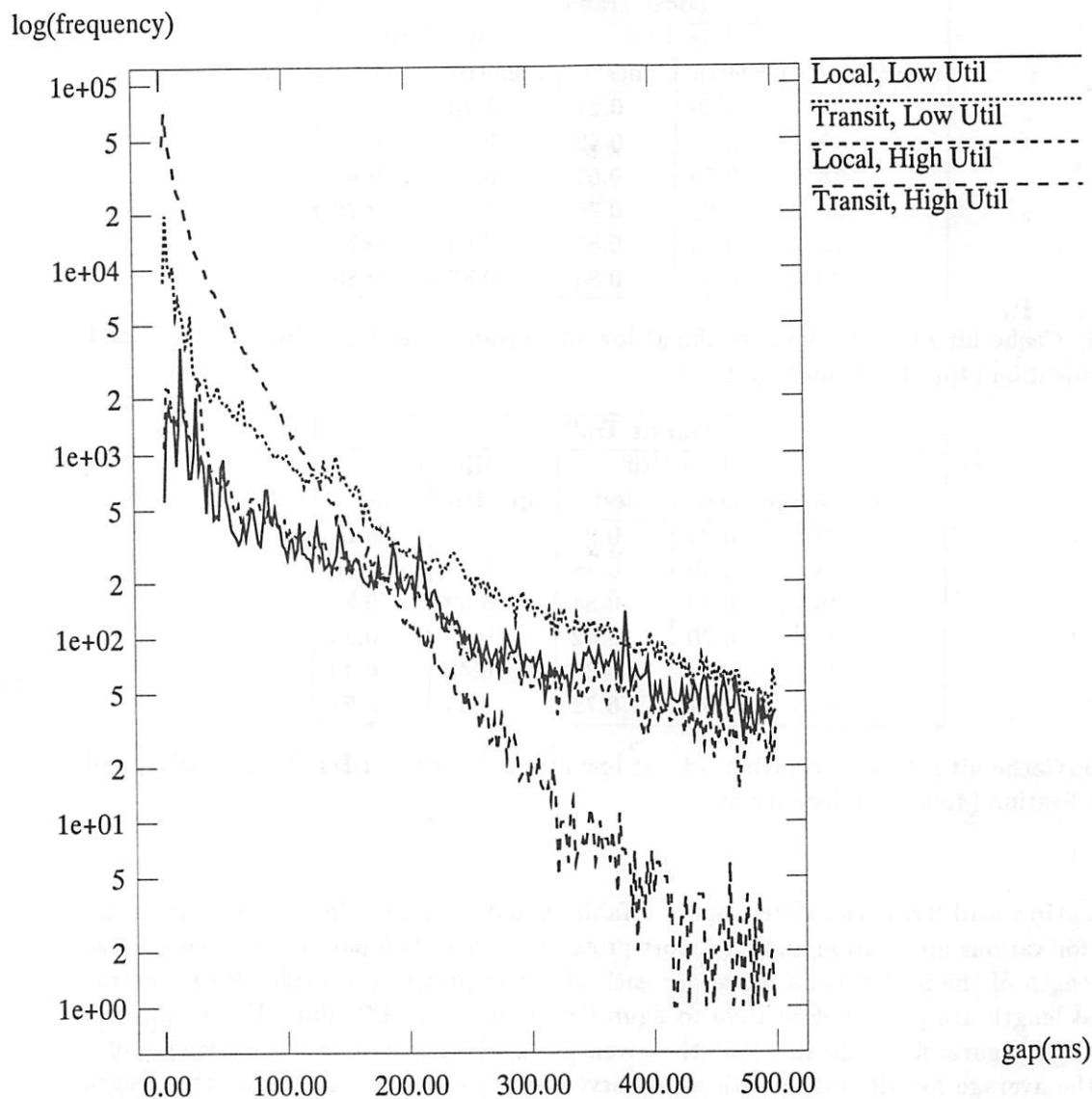


Figure 9: Interarrival time distribution for UDP traffic.

constant with a mean of 9.2 cars and standard deviation of 0.23. Table 7 shows mean intercar and intertrain gaps for local and transit traffic at high and low utilizations. Table 8 shows maximum allowed intercar gaps. Differences in MAIG due to both type of traffic and utilization are apparent here. It is interesting to note that figures for transit traffic are quite variable, possibly more subject to the influence of FTP traffic (a characteristic shared by local traffic at low utilization but not reflected in samples here) or to the influence of occasional interference from other regional networks connected to different parts of the backbone.

Local Traffic				
cache size	Low Util		High Util	
	prefetch	dest	prefetch	dest
2	0.29	0.21	0.20	0.22
8	0.62	0.49	0.47	0.38
16	0.76	0.67	0.66	0.57
24	0.82	0.75	0.77	0.70
32	0.85	0.80	0.83	0.78
40	0.88	0.84	0.87	0.83

Table 5: Cache hit ratios for local traffic at low utilization (2am EDT, locality 28%) and high utilization (4pm EDT, locality 19%).

Transit Traffic				
cache size	Low Util		High Util	
	prefetch	dest	prefetch	dest
2	0.23	0.20	0.10	0.15
8	0.49	0.38	0.24	0.26
16	0.63	0.54	0.32	0.32
24	0.70	0.62	0.38	0.38
32	0.74	0.68	0.46	0.44
40	0.77	0.72	0.54	0.51

Table 6: Cache hit ratios for transit traffic at low utilization (2am EDT, locality 20%) and high utilization (4pm EDT locality 8%).

**Application and Transport Protocols** Tables 9 and 10 present intercar gap and train length for various application and transport protocols. These two parameters characterize the strength of the packet train model for each of the respective protocols. Both intercar gap and length are presented relative to figures computed for all traffic. For example, a train length figure of 3 indicates that the given protocol exhibits mean train length of 3 times the average for all traffic; with an observed average of 9 for all traffic, this length would be 27 cars. Deviation is the standard deviation among the relative figures for each sample.

Upon examining Tables 9 and 10, the most striking results are those for FTP. FTP exhibits a relatively low intercar gap and an astoundingly high relative train length. The high deviation reported here is the result of unusual FTP behavior (with regard to most samples) of one of the samples.

The low intercar gap associated with the network management protocols (SGMP and SNMP) is somewhat puzzling at first glance. This relatively small gap can be explained by the fact that most network management packets traversing the backbone are in fact destined to packet switches *within* the backbone, resulting in a fewer number of hops and less transmission delay than if the packets had to exit the backbone and traverse a destination regional network as well.

Intercar gap for domain nameserver traffic is relatively high. This protocol is a request-

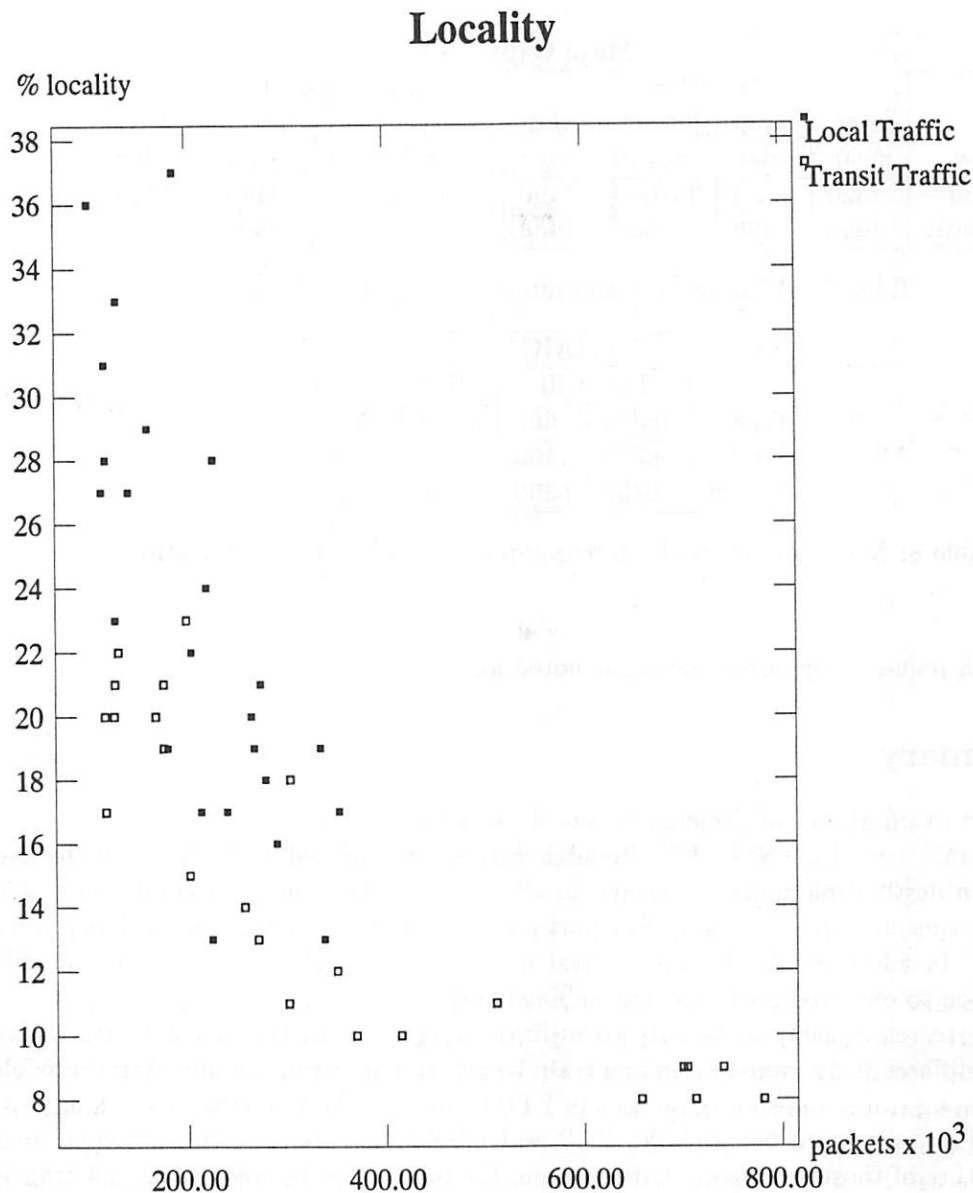


Figure 10: Locality vs. utilization.

response protocol. Name queries entering the backbone are likely to be headed toward a host responsible for serving a large number of hosts, where significant host load could increase the response time. For all but transit traffic at low utilization, mean train length for this protocol is exceptionally low; this fact can also be attributed to the request-response nature of domain nameserver traffic.

The contrast between TCP and UDP transport protocols is not surprising. In all cases, TCP has a mean intercar gap less than that for UDP (and that for combined traffic) and an expected train length much greater than the length for UDP traffic. These differences are easily explained by the characteristics of the protocols; TCP is a sliding window protocol much more likely to send multiple packets into a network than is UDP, a datagram protocol. Most application protocols (e.g., the domain nameserver) using UDP are transaction



Mean Gaps								
type	Low Util				High Util			
	Intercar mean	Gap dev	Intertrain mean	Gap dev	Intercar mean	Gap dev	Intertrain mean	Gap dev
local	320	34	33100	1400	140	10	13140	1100
transit	1050	290	43800	7450	290	90	19800	3600

Table 7: Mean intercar and intertrain gaps for all traffic.

MAIG				
type	Low Util		High Util	
	mean	dev	mean	dev
local	6350	1100	1040	$\approx 0$
transit	16100	6300	3100	570

Table 8: Maximum allowed intercar gap mean and standard deviation.

oriented with request-response behavior as noted above.

## 6 Summary

Through the examination of prefetch versus destination caching, this study has verified the packet train model on NSFNET. Prefetch caching has proved to be a more effective strategy than destination address cacheing in all practical situations. For combined traffic as well as for specific application and transport protocols, the packet train model has proved to be valid. In addition, the Poisson arrival model has been shown to be a reasonable approximation to observed patterns for combined traffic.

The inverse relationship of locality to utilization agrees with the original MIT study [7], and the differences in intercar gap and train length among various application protocols (such as the long trains and short gaps seen in FTP traffic and the short trains and long gaps seen in SMTP traffic) and between the TCP and UDP transport protocols are consistent with the nature of those protocols. Furthermore, the distinction between local and transit traffic is clearly reflected in the chosen maximum allowed intercar gap.

### 6.1 Directions for Further Work

The spikes near 6 ms in the interarrival time curves deserve further study. Unfortunately, the 1 ms resolution of the RT's clock prevents a more detailed view of interarrival time in this range. Higher clock resolution would lead to a more accurate picture and hopefully better determination of the spike's cause. The spike should move closer to 0 ms once the speed matching boxes are removed and as the packet switching delays are reduced. Furthermore, it would be interesting to separate local traffic into traffic entering the backbone versus traffic exiting the backbone. Differences in the resulting interarrival gap curves could show the affect on traffic of the backbone itself.

Study of IP address reference patterns such as that found in [6] would be useful as well. It is possible that the least recently used replacement policy does not provide optimum use

Low Utilization								
protocol	Local Traffic				Transit Traffic			
	Intercar	Gap	Train	Length	Intercar	Gap	Train	Length
	mean	dev	mean	dev	mean	dev	mean	dev
Domain	2.41	0.52	0.20	0.02	1.66	0.21	0.75	0.07
FTP	0.65	0.18	17.75	11.23	0.29	0.18	23.08	11.30
Login	0.83	0.24	1.38	1.01	0.73	0.29	1.23	0.41
Netman	0.55	0.06	2.00	0.26	0.37	0.24	0.71	0.16
SMTP	1.40	0.19	0.80	0.02	0.96	0.16	1.13	0.12
TCP	0.85	0.03	2.15	0.34	0.55	0.03	2.03	0.08
UDP	1.45	0.25	0.31	0.02	1.90	0.23	0.69	0.04

Table 9: Relative intercar gap and train length under low utilization.

High Utilization								
protocol	Local Traffic				Transit Traffic			
	Intercar	Gap	Train	Length	Intercar	Gap	Train	Length
	mean	dev	mean	dev	mean	dev	mean	dev
Domain	1.25	0.05	0.18	0.01	1.57	0.29	0.39	0.10
FTP	0.64	0.16	5.64	0.50	0.48	0.05	6.80	1.74
Login	1.06	0.03	1.50	0.14	1.02	0.14	2.12	0.34
Netman	0.92	0.05	0.94	0.13	1.21	0.39	0.69	0.10
SMTP	1.21	0.04	0.60	0.05	1.38	0.44	0.89	0.05
TCP	0.90	0.01	1.40	0.09	0.87	0.06	1.86	0.16
UDP	1.04	0.05	0.27	0.02	1.47	0.20	0.37	0.09

Table 10: Relative intercar gap and train length under high utilization.

of the address cache. A study of autocorrelation of IP addresses using different lag times or packet counts could provide insight into this problem.

Varying the MAIG selection policy could provide new insights about the traffic patterns. In particular, choosing a MAIG value as a multiple of the observed mean interarrival gap could prove to be a more effective strategy than the 90 percentile policy used now.

An analysis of more protocols would be useful. Remote filesystem protocols, windowing protocols, and transaction-oriented transport protocols such as VMTP, in particular, would be interesting to fit to the packet train model. The currently low level of traffic on NSFNET for these protocols prevents this analysis.

Running a tool such as IPwatch [8] on the backbone would allow real-time monitoring of train locality. Presumably, this tool could be extended to provide a dynamic view of train parameters and a better idea of the effect of various application protocols on these characteristics.

## 7 Acknowledgements

I'd like to thank Professor Satish Tripathi of the University of Maryland for his advice concerning the final draft of this paper and his support throughout the project. Ólafur Gudmundsson of the University of Maryland provided countless ideas for sources of delay which could cause the "spike." Jacob Rekhter of IBM gave valuable comments at both the inception and conclusion of the project. Monitoring of NSFNET was made possible by Dave Katz of MERIT; Dave set up the monitor RT at the Ann Arbor NSS, provided a modified kernel and NNStat software, and answered many questions about NSFNET during the project. Hans-Werner Braun of MERIT gave administrative approval. Nguyen Hien of IBM gave advice concerning the final draft of this paper and handled the extensive paperwork needed to provide both equipment and financial support from IBM.

## References

- [1] Robert T. Braden and Annette L. DeSchon. NNStat: Internet statistics collection package – introduction and user guide. Technical report, University of Southern California / Information Sciences Institute, Marina del Rey, CA, November 1988.
- [2] J. Case, M. Fedor, M. Schoffstall, and C. Davin. A simple network management protocol (SNMP). RFC 1098, Network Information Center, April 1989.
- [3] J. Davin, J. Case, M. Fedor, and M. Schoffstall. A simple gateway monitoring protocol (SGMP). RFC 1028, Network Information Center, November 1987.
- [4] D. C. Feldmeier. Empirical analysis of a token ring network. Laboratory for Computer Science Technical Memo 254, Massachusetts Institute of Technology, Cambridge, MA, January 1984.
- [5] Riccardo Gusella. The analysis of diskless workstation traffic on an Ethernet. Computer Science Division Technical Report UCB/CSD 87/379, University of California, Berkeley, CA, November 1987.
- [6] Raj Jain. Characteristics of destination address locality in computer networks: A comparison of caching schemes. Technical Report 592, Digital Equipment Corporation, Littleton, MA, February 1989.
- [7] Raj Jain and Shawn A. Routhier. Packet trains – measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communications*, SAC 4:986–995, September 1986.
- [8] Mark J. Lorence and M. Satyanarayanan. IPwatch: a tool for monitoring network locality. In *Proceedings 4th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, September 1988.
- [9] K. McCloghrie and M. T. Rose. Management information base for network management of TCP/IP-based internets. RFC 1065, Network Information Center, April 1988.
- [10] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: an efficient mechanism for user-level network code. In *Proceedings ACM SIGOPS*, November 1987.

- [11] John Nagle. On packet switches with infinite storage. RFC 970, Network Information Center, December 1985.
- [12] J. Postel. User datagram protocol. RFC 895, Network Information Center, April 1984.
- [13] J. Postel and J. Reynolds. Telnet protocol specification. RFC 854, Network Information Center, May 1983.
- [14] J. Postel and J. Reynolds. File transfer protocol (FTP). RFC 959, Network Information Center, October 1985.
- [15] Jon Postel. Internet protocol. RFC 791, Network Information Center, September 1981.
- [16] Jon Postel. Transmission control protocol. RFC 793, Network Information Center, September 1981.
- [17] Jon Postel. Simple mail transfer protocol. RFC 821, Network Information Center, August 1982.
- [18] Jacob Rekhter, August 1988. Personal communication.
- [19] Cheng Song and Lawrence H. Landweber. Optimizing bulk data transfer performance: A packet train approach. In *Proceedings ACM SIGCOMM*, 1988.



**Steve Heimlich**  
*U of Maryland*

Steve Heimlich graduated from Dartmouth College in 1987 with a B.A. in computer science. He received his M.S. in computer science from the the University of Maryland at College Park in 1989. He is currently employed at the IBM T.J. Watson Research Center where he continues to work on NSFNET and related TCP/IP projects.





# Pseudo-Network Drivers and Virtual Networks

S.M. Bellovin\*

smb@ulysses.att.com

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974

## ABSTRACT

Many operating systems have long had *pseudo-teletypes*, inter-process communication channels that provide terminal semantics on one end, and a smart server program on the other. We describe an analogous concept, *pseudo-network* drivers. One end of the driver appears to be a real network device, with the appropriate interface and semantics; data written to it goes to a program, however, rather than to a physical medium. Using this and some auxiliary mechanisms, we present a variety of applications, including system test, network monitoring, dial-up TCP/IP, and ways to both improve and subvert network security. Most notably, we show how pseudo-network devices can be used to create *virtual networks* and to provide encrypted communications capability. We describe two implementations, one using a conventional driver for socket-based systems, and one using stream pipes for System V.

## 1. INTRODUCTION

Many operating systems have long had *pseudo-teletypes*, inter-process communication channels that provide terminal semantics on one end, and a smart server program on the other. In the same vein, we have implemented a *pseudo-network* driver. To the kernel, and in particular to IP, it appears to be a device; instead of transmitting the bits over a wire, the output packets are sent to a program. Similarly, packets written by the program are delivered to the network input handlers, exactly as if they were received over a real device. The general flow of control is shown in Figure 1.

IP (or another network protocol) hands packets to the bottom half of *Pnet*; the top half of the driver passes them to a server program, which can communicate with other servers. Similarly, the server can generate packets and pass them to the driver; these are in turn sent to IP.

There are two general implementation techniques available. For socket-based systems, such as SunOS and 4.3bsd, we have implemented a standard network device driver; a detailed description of the driver is given below. For *stream*<sup>[Ritc84]</sup> implementations of TCP/IP, a simple stream pipe may suffice, possibly with no kernel changes whatsoever; again, details are given below.

---

\* Author's address: Steven M. Bellovin, Room 3C-536B, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974.

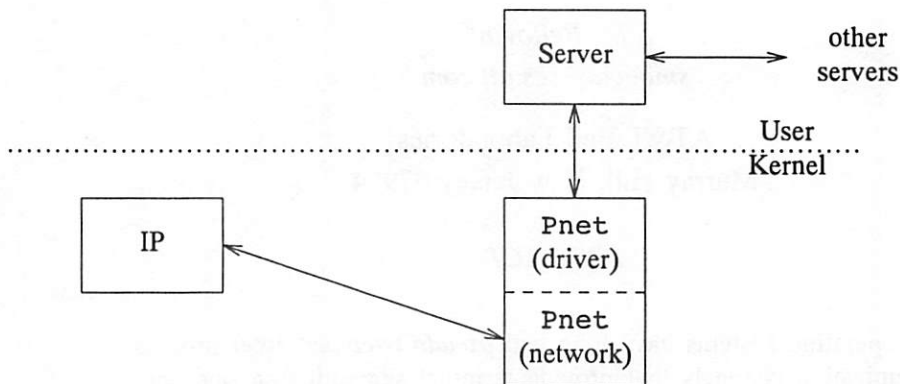


Figure 1. The Pseudo-Network Driver

Although the primary focus of the driver is TCP/IP,<sup>[Fein85, Come88]</sup> the socket version is actually quite general; it can handle any address families supported by the rest of the kernel. It has been tested on SunOS 4.0.1 and 4.0.3; with minor changes, it should run on 4.2bsd, 4.3bsd, and other related operating systems.

## 2. RE-INJECTION TECHNIQUES AND ISOLATED INTERFACES

A number of uses for `Pnet` involve *re-injecting* a transformed packet into the kernel for further processing. For example, the packet could be encrypted, repackaged with a new IP header and a protocol number indicating encryption, and sent on its way. Before discussing `Pnet` proper, it is worth examining possible mechanisms for re-injection; it is not trivial to implement, but is quite necessary.

The first, and most obvious way, is to build a new packet, and simply `write()` it to the `Pnet` device, under the assumption that IP will then forward it to the proper destination. However, many IP modules will *not* forward packets, either for security reasons or because forwarding packets is the business of gateways, not hosts.<sup>[Brad89]</sup>

For socket-based implementations, a second approach is to create a raw IP socket, and use it to re-inject the packets. Unfortunately, while that mechanism is suitable for transmitting the encrypted packets, it fails on decryption. Decrypted packets — received by a user-level process bound to that IP protocol number — should carry the IP source address of the original sender; the raw IP socket interface insists that packets carry authentic local source addresses. While it may be possible to kludge around this requirement, a cleaner solution can be obtained by implementing a new raw protocol in the Internet address family; this protocol would permit specification of an arbitrary IP header.<sup>1</sup>

We have opted to implement a variant of this mechanism. Rather than create a separate interface solely for packet re-injection, we have overloaded the address family field used by `pnetwrite`. As noted, these packets are passed directly to the IP output routine, rather than the input routine. This interface must be used with great care. Only minimal checks are done, to guard against kernel panics. No attempt is made to provide standard packet input processing, such as checksum validation, time-to-live counter decrementing, or option processing. More seriously, the packet is not checked to see if it is destined for this host. If it is, when the real driver receives the packet, it must pass it to IP's input routine. Of course, if the packet was destined for `Pnet`'s local address, it will be delivered again to the server, possibly causing a loop. `Pnet` broadcast packets are a particularly nasty case of this.

Implementing re-injection is harder for stream implementations. The only path into IP is the transport protocols' interface; for these, IP expects to fill in the source address, etc. Some sort of raw channel is needed; this might require changes to IP.

An alternative to packet re-injection is to implement *interface isolation*. If an interface is marked as *isolated* (presumably via `ifconfig`), packets from it are not forwarded. Thus, packets arriving via the `Pnet` driver could be forwarded, while packets arriving on an external link would not. Obviously, source routing would be disabled for isolated interfaces also. A final aspect of interface isolation, possibly controlled by a different bit, is to accept packets arriving on an isolated interface if and only if they are destined for the machine's IP address on that interface. That is, we do not permit the implicit forwarding to an alternate address associated with another network interface on the gateway.<sup>2</sup>

For some purposes, a simple isolation bit is insufficient; one would need isolation groups that define allowable forwarding patterns. Finally, one could escalate to full address screening,<sup>[Mogu89]</sup> though if encryption is universally performed that is probably not necessary.

### 3. APPLICATIONS

The `Pnet` driver has many uses, ranging from the trivial to the complex. A few are discussed below. We have implemented some of these, and plan to implement others.

#### 3.1 System Test

It is often difficult to test protocol implementations. The usual approach is to use sophisticated network monitors to observe the traffic and to create test packets. Such techniques, though, are expensive and often uncertain — fast hosts can easily overrun some network monitors. `Pnet`, though, makes life much easier — a host program can catch or generate all test packets.

Care must be taken when emulating a protocol in a program; some features are more difficult to emulate than others. During development of `Pnet`, we ran into trouble with fragmented ICMP ECHO packets; generating proper replies required receipt and reassembly of all

1. Some people may object that allowing a host process to impersonate an IP address is a security risk. First, this facility is only available to `root`; a rogue super-user has easier ways to spoof IP addresses. Second, the very existence of `Pnet` allows injection of packets with arbitrary addresses. Finally, as shown elsewhere,<sup>[Bell89]</sup> using an IP address for authorization is very unsafe in any event.
2. This does not necessarily provide enough security for the gateway machine. ICMP packets can have a global effect, regardless of the destination address used.

fragments.

### 3.2 Netspy

The `Pnet` driver can be used to monitor conversations. A routing entry can be constructed to direct traffic for a particular destination to the `Pnet` driver. After examination, the packet can be re-injected, adding IP Loose Source routing to carry the packet to the next hop.

The potential for abuse of this capability is, of course, obvious.

### 3.3 Non-IP Relays

In some environments, it is necessary to send IP packets over media for which IP drivers do not exist. `Pnet` provides a simple mechanism for accomplishing this; a program can retrieve the IP packets via the `Pnet` driver, encapsulate them for some other protocol, and transmit them to the far end.

This can be also be done in some situations where one side implements an IP driver directly. For example, some implementations of Datakit<sup>®</sup> VCS support contain an IP interface, while others do not. The latter can use `Pnet` to transmit packets to and from IP; at the far end of the Datakit VCS circuit, IP can handle them directly.

### 3.4 Replacing SLIP

The conventional mechanism for sending IP packets over tty lines — SLIP, or Serial Line IP<sup>[Romk88]</sup> — requires oddball code in the kernel. A line discipline is used for framing, which is reasonable enough; however, some implementations require a dummy process to linger to keep the line open, or some mechanism to prevent the normal close operations from taking place. Furthermore, dial-up SLIP operation is awkward, though it has been done.<sup>[Lanz89]</sup> All of that can be bypassed using `Pnet`. A single process can handle packets for all of the SLIP destinations; it can make calls as needed, transmit and receive data, etc. To be sure, a line discipline may be needed in any event, to buffer the incoming characters and avoid the need to wake up the SLIP daemon each time, but much of the complexity could be eliminated.

A `Pnet` implementation has the side-effect that all of the SLIP destinations would share the same IP network number. This is probably a good idea — using an entire network for each point-to-point link is wasteful, though presumably one could subnet a class C network and use it for 64 SLIP links<sup>3</sup> — but it requires good routing protocols to handle the point-to-point connections. The IP model normally requires that an interface driver be able to reach every connected host directly; this is often not the case with SLIP.

### 3.5 Bypassing Security Controls

The `Pnet` driver can also be used to implement a bypass for some common security controls. Assume, for example, a paranoid gateway that was configured to allow only electronic mail traffic; this would be configured to accept TCP packets with a source or destination port of 25, and to reject all others.<sup>[Mogu89]</sup> Two co-operating parties could set up a TCP circuit between `Pnet` servers, and simply assign one end to port 25. Assuming suitable routing information were exchanged, each end would have access to the other's IP networks.

---

3. At least 2 bits must be used for every subnet, as the host addresses 0 and -1 are still reserved.

Obviously, in this sort of situation two parties who merely wished to leak information could do so rather more simply. The point is that *Pnet* allows IP-level access, and is thus far more damaging.

### 3.6 More Reliable Datagrams

In the current congested Internet environment, datagram services are hard to use. Too many packets are dropped or delayed, leading to excessive retries and/or congestion.<sup>[Nowi89]</sup> If a TCP-based relay process is used with *Pnet*, application-level retry timers can be turned off, and advantage can be taken of recent TCP performance improvements.<sup>[Karn87, Jaco88]</sup> Similarly, if the underlying network is prone to data corruption, this mechanism is useful when using systems that turn off UDP checksumming.

If this strategy is adopted, great care must be taken if application-level retry timers are still used. TCP segments can be delayed or lost as easily as UDP packets; however, since TCP will retransmit on its own, it is highly undesirable for the application to do so as well. Application-level retransmissions will simply generate extra load; they will not provide better service.

## 4. VIRTUAL NETWORKS

There are a number of protocols, typically broadcast-based ones, that operate properly only within a single IP network. If the machines that wish to run such a protocol are geographically dispersed, it may not be feasible to connect them to the same net. Using *Pnet*, though, this can be accomplished reasonably easily: a server could declare the interface to be a broadcast network, and transmit broadcast packets to all appropriate destinations. This is an example of a *virtual network*. While there is an obvious efficiency loss in broadcast virtual networks, the gain in functionality may make it worthwhile for some applications.

Virtual networks have other uses as well. For example, consider the case of a large corporation with many internal TCP/IP networks, and a single gateway to the Internet. It may be desirable to allow a very few selected hosts access to the Internet through the gateway; most, though, would be blocked for security reasons. The selected hosts and the gateway could form a virtual network; only its address would be advertised to the outside world.

A more general way to phrase this is that virtual networks allow for routing and control independent of the physical topology. This can be used to implement many different useful schemes, including “roamer hosts”.

Virtual network packets may be carried by TCP, UDP, or IP. If UDP is used, checksumming should be turned off for that connection; it represents needless expense, as the encapsulated packet will undergo further validity checking when delivered to its ultimate destination.

### 4.1 An Encrypted Virtual Network

Perhaps the most interesting use of *Pnet* is to implement encryption, access control, and authentication mechanisms. We shall spend some time on a detailed description of just such a system; it is currently under development. Since ours is loosely modeled on the *Blacker Front End*<sup>[BFE, Mund87]</sup> but is much less secure, we dub it *Greyer*. There are two principal uses for *Greyer*: providing end-to-end encryption between a pair of hosts communicating over an insecure network, and providing network-level encryption between a pair of gateways, each of which is protecting a group of naive hosts. We will consider each design in turn.



At first blush, providing end-to-end encryption is simple. Create a virtual network, as described above. When a host wishes to make a secure call to a destination, it uses the destination network's address on the virtual network. All of the packets are thus delivered to the `Pnet` server, which encrypts them and sends them along. These servers have addresses on the insecure physical network. The destination server receives the packet, decrypts it, and writes it on the `Pnet` device; in the kernel, the packet is recognized as destined for the local host, and is delivered to the application in the usual fashion.

There is a catch, however. One of the benefits of encryption is the implied authentication it provides. Applications which believe they are conversing over the secure virtual net may quite reasonably extend much greater trust. Unfortunately, packets with a virtual net destination address may be delivered to a host over its physical network interface; these packets have not been validated in any way. They will nevertheless be accepted.

The easiest solution is the interface isolation mechanism described earlier. Note that we must isolate the physical network, not the virtual one. That is, we will accept packets over the virtual interface for either address; we do not wish to accept packets for the presumed-secure virtual address over the physical link. If the host has more than one physical address, this solution is too simplistic; it may be necessary to use isolation groups.

Some may object that this is not a real problem. After all, even though forged packets to the protected address may be sent via the physical network, replies will be sent via the virtual network, and hence will be encrypted. Unfortunately, there are ways to attack hosts that rely on IP addresses for authentication, even if responses are not heard.<sup>[Bell89, Morr85]</sup> More simply, IP source routing could be used, thereby forcing the target host to reply via the same insecure path.

#### Gateways and Greyer

Gateways using `Greyer` may require interface isolation as well. For inbound traffic, the rationale is simple: we do not wish unauthorized packets to enter the protected subnet. If the interface were not isolated, an enemy could simply use the physical network address of a target host.

Outbound traffic may need to be restricted also. In a classified environment, for example, individual users may not select the data transmission mode; that is up to the administrator. It is thus necessary to guard against internal traffic being routed directly to the external interface. On the other hand, we cannot simply turn off packet-forwarding, or we would have no way to deliver outbound packets to the `Pnet` server.

Our model, then is this: hosts behind the `Greyer` gateway forward their packets to it to reach a remote secure host. On the gateway, routing table entries specify that the next hop is on the virtual net; this forces the packets to be delivered to the `Pnet` server for encryption. The packets are encrypted and encapsulated, and transmitted over the insecure network to a remote server. It must then decrypt them and hand them back to IP. We may use re-injection; if the host will permit packet-forwarding from the `Pnet` interface, a `write()` over the `Pnet` device will serve.

As noted, encryption provides implied authentication. It also provides authorization: the key distribution center may, at its option, decline to issue a key for a conversation deemed administratively prohibited. In fact, the `Greyer` mechanisms could simply be used for authorization without bothering with transmitting the encrypted text at all, as in the *Visa* protocols.<sup>[Estr89]</sup> There are obvious risks of address forgery here, of course.

### Encapsulation for Greyer

There are two issues to consider when deciding how to encapsulate Greyer packets for transmission over the insecure network: how should session key information be distributed, and what transport mechanism should be used? The two questions are related.

First, we assume that the Greyer server will not have keys for each possible destination; rather, it will use something like Needham-Schroeder<sup>[Need78, Denn81, Need87]</sup> or Kerberos<sup>[Ste88]</sup> to obtain a session key. It is therefore necessary to transmit this session key to the remote Greyer server. If TCP is used as the transport mechanism, the solution is obvious: send the session key at the start of each connection. If a key expires, the connection may be torn down and a new one constructed.

If, on the other hand, a datagram mechanism is used (either UDP or a new IP protocol type), the problem is a bit harder. One possibility is to send a special packet containing the key to the remote Greyer server; depending on the reliability of the underlying network, it may be desirable to await an acknowledgement before transmitting any packets that use the key. More likely, we will use the SP3 protocol from SDNS.<sup>[SP3]</sup>

A final possibility is to include the encrypted key in each packet. This preserves the stateless nature of IP gateways, at the obvious cost in bandwidth. The exact choice depends heavily on the characteristics of the physical network; we will address this question further when Greyer is implemented.

## 5. SOCKET IMPLEMENTATION DETAILS AND ALTERNATIVES

The socket `Pnet` driver consists of two distinct halves, a network driver and a character device driver. Each contains the usual entry points: `attach`, `output`, and `ioctl` for the network driver, and `open`, `close`, `read`, `write`, `ioctl`, and `select` for the character driver. We describe each half in turn.

The network output routine (`pnoutput`) is quite straight-forward. If the character half of the driver is not open, packets are rejected with code `ENETDOWN`. Otherwise, the packet is queued for the server program. A header containing the destination address is prepended to the packet, in the form of a single `struct sockaddr`. It is important that the server program use this address to determine the header, rather than looking at the packet header; to do otherwise would require that it duplicate most of the functions of IP. If the program's input queue is full, the packet is discarded and `ENOBUFS` is returned to the caller. No attempt is made to loop back packets destined for a local address; that is left to the server.

The rest of the network driver half is comparatively trivial. One, perhaps incorrect, decision: if the interface is turned off via `SIOCSIFFLAGS`, the server program is sent an EOF message.

The character driver is a bit more complex. `Pnetread` blocks until data has been enqueued by `pnoutput`; if `FASYNC` mode has been selected, it returns an error code instead if the queue is empty.

`Pnetwrite` is more problematic for several reasons. First, it cannot accept just a raw packet; it needs address family information in order to route the packet to the proper protocol. While a simple `short` would suffice, the current driver requires a full `struct sockaddr`; this simplifies use of the same data structures for the input and output halves of the program. The other fields in this structure are currently unused, though that may change in the future.

A second complication is the need to *re-inject* packets into the system, as described above. If the high-order bit of the address family is on, the packet is passed to the *output* routine of that protocol, rather than the input routine. Currently, only `AF_INET` is supported for this option.

Finally, it is not obvious how to block if the protocol input queue is full. It is easy enough for the server process to sleep; however, there is no “interrupt routine” to awaken it when the queue drains. Accordingly, a timer routine is used to poll the queue status.

`Pnetselect` has a similar problem; additionally, since it lacks information on which protocol input queue is desired, it cannot assert definitively that space is available. As a heuristic, it queries the status of the last queue to which a `write()` was attempted.

`Pnetioctl` permits the server to set the `IFF_BROADCAST` and `IFF_POINTOPOINT` flags for the interface; since the driver has no way of knowing the intended use of the interface, it cannot make a default choice. Additionally, the server can set and reset `IFF_UP`; while this flag can be set via `SIOCSIFFLAGS`, use of that `ioctl()` is restricted to `root`.

It is also possible for the server to change the maximum transmission unit (MTU) allowed for the interface. If another network medium is used to relay `Pnet` packets, the MTU for the `Pnet` interface should be set to the MTU of the medium minus any required headers, to avoid fragmentation.

#### Rejected Alternatives

An alternative implementation technique would have been to replace the character driver with a new `socket` address family. That would have allowed use of `sendto()` and `recvfrom()` system calls to pass the auxiliary address, rather than requiring a prepended header. Similarly, much of the existing code for `socket` input/output could be used, rather than writing new routines. This approach turned out to be infeasible for several reasons.

The first is simply a question of packaging. As some systems are distributed, it is much easier to add new device drivers than to add new address families. There is no accessible table to configure the `domain` structure for a new address family, nor are there vacant entries in the address family name space. While some dormant entry could be reused, this seemed unwise. Nor is it possible to add additional entries to a binary system; there are several routines, and one table, that “know” how many address families there are.<sup>4</sup>

A second reason is the permission structure. It was useful to permit non-`root` users to access this facility, at least during testing; this is very easily accomplished via the file system’s permission mechanisms. Doing the same for a `socket` family would have been awkward.

Finally, the device driver interface is much more standardized across releases than is the `socket` interface, and much more documentation exists for it.

It should be noted parenthetically that although modifying distributed source code is not *a priori* a bad idea, it is often infeasible. Source code distributions are sometimes not as current as binary-only distributions, and not everyone is licensed to receive source code.

4. Amusingly enough, the SunOS 4.0 distribution does not have `AF_MAX` set high enough for all of the address families named in `socket.h`.

### 5.1 Performance of Socket-Based Pnet

Obviously, performance is a concern when packets are copied to and from user level an extra time before being transmitted. To measure the performance of the socket implementation, we employed a modified version of `ping(8)`. This version transmitted a new ICMP ECHO packet immediately upon receipt of the response to the previous packet; it also printed the total elapsed time for the packet sequence. Employing this technique, rather than measuring the per-packet round-trip time, allowed us to avoid problems with the coarse granularity of the system clock. In the actual test, between a Sun 3/60 and a Sun 3/75, we measured performance at user-data lengths ranging from 0 to 1300 bytes. Each measurement consisted of 100 ICMP packets; we repeated each test 100 times. The goal was to account for both the per-packet and per-byte overhead. UDP was used as the transport mechanism, with checksumming turned off (the SunOS default). Each test was repeated several times. Note that the timing represents four copy operations on each byte: when sending the ECHO packet, when it is received on the target machine, when the response packet is sent, and when it is received.

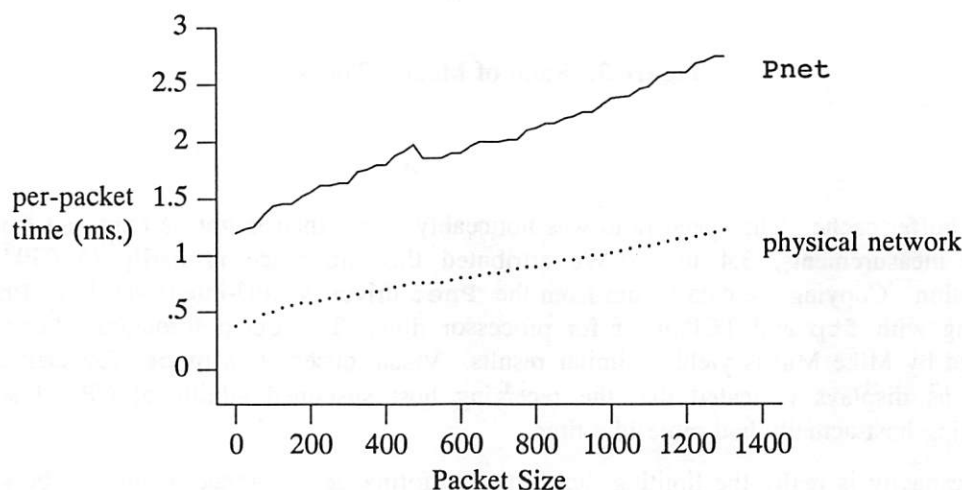
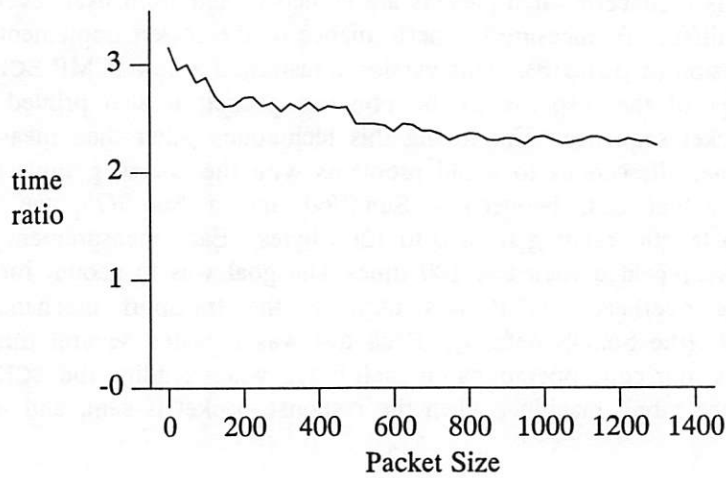


Figure 2. Median ICMP ECHO Time

Figure 2 shows the median times in milliseconds for each packet size. There is a glitch in the graph at around 500 bytes; this is most likely due to buffer allocation strategies. Packets of more than 512 bytes — counting the IP and ICMP headers, in this case — are copied into a single *mbuf cluster*, rather than a chain of *mbufs*.

A second graph, Figure 3, shows the ratio of the times. It appeared that `Pnet` performed at one half to one third the speed of the raw underlying network. To validate this, we used `ftp(1)` to copy a large file to `/dev/null`, after ensuring that the entire file was in the





**Figure 3.** Ratio of Median Times

sender's buffer cache. The speed ratio was noticeably worse than might be expected from the previous measurements, 3.4 to 1. We attributed this difference primarily to CPU time consumption. Copying the data to and from the `Pnet` driver is CPU-intensive; thus, `Pnet` is competing with `ftp` and TCP itself for processor time. The `ttcp` throughput benchmark developed by Mike Muuss yielded similar results. Visual observation of `perfmeter` during `ttcp` runs displays indicated that the receiving host sustained additional CPU load; the transmitting host actually had more idle time.

If CPU capacity is really the limiting factor, the performance difference would not be seen if there was extra CPU capacity available. Looked at another way, we can make available more CPU time per packet by slowing down the interarrival rate. This was most easily accomplished by running similar tests across a long-haul link, in this case between Murray Hill, New Jersey, and Allentown, Pennsylvania. IP access between the two sites is via a 1.344M bps point-to-point link; additionally, several other local area networks and gateways intervene at each end.

The raw throughput graph is shown in Figure 4; the speed ratio is shown in Figure 5. Performance is a bit more variable, due to the vagaries of the shared link; as can be seen, though, the shapes of the two sets of graphs correspond nicely. The `Pnet` link is only about 1.1 times slower than the direct link in this case. A complication arose because of packet loss on the link; given the design of the test program, each dropped packet caused a one-second timeout before the next ICMP packet was sent. We adjusted for this by subtracting one second from the total time for each such packet.

The overhead of `Pnet` should be relatively constant for a given packet size, regardless of the link speed. Figure 6 shows the difference in throughput for both sets of tests; as can be seen,



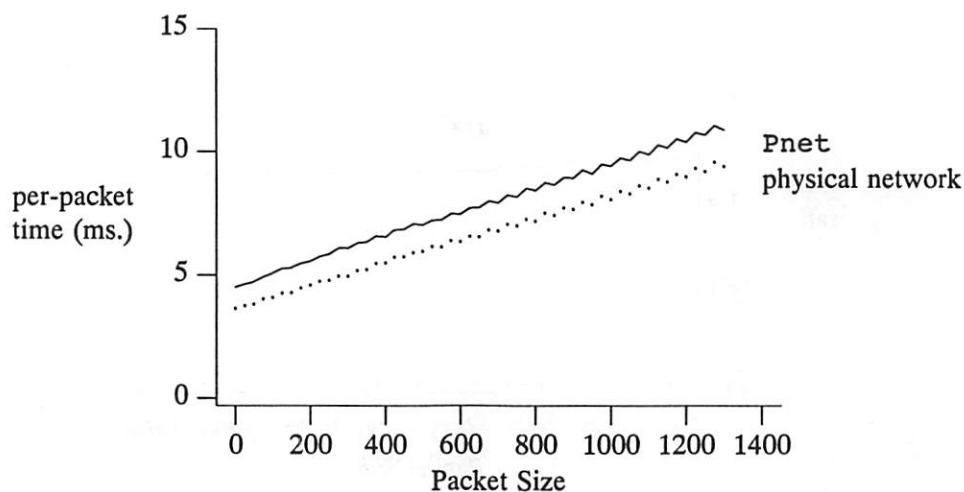


Figure 4. Median ICMP ECHO Time — Long Haul

the two graphs are quite similar.

Finally, the same `ftp` and `ttcp` tests were run; throughput for `Pnet` was essentially the same as on the physical network.

## 6. A STREAM VERSION OF `Pnet`

On a system with a good stream implementation of TCP/IP, there is no need for a `Pnet` driver. The native drivers can be used instead, for all of the applications described above. The mechanism is quite simple: create a stream pipe, and link one end of it to IP. Issue the appropriate configuration `ioctl()` calls (i.e., to inform IP of the network number and IP address), and the stream will be treated the same as any other device driver. For conventional devices, this configuration process is typically table-driven. Since `Pnet` devices are dynamically created, a table is not usable; instead, the `Pnet` server must handle the process manually.

Shutting down a pipe-based `Pnet` driver is often difficult. Shutdown may be disorderly; one or both ends of the pipe may be closed before IP's `close` routine is called. It is therefore vital to detect `M_HANGUP` messages traveling upstream. Another crucial detail is whether IP is prepared to delete the interface control structures. In some versions of stream TCP/IP, much of the rest of the networking code is unprepared to deal with the possibility of such deletions. For example, route table entries often point to the per-interface structure; if these are not cleaned up, problems can occur. In fact, some early implementations of SLIP for 4.2bsd were known to crash when the interface was deleted.

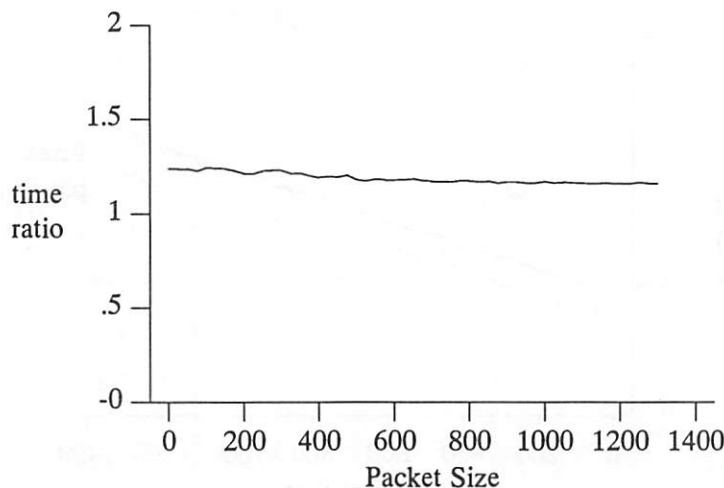


Figure 5. Ratio of Median Times — Long Haul

The 9th Edition implementation of stream TCP/IP deals well enough with shutdowns; however, the IP destination address is not passed downstream along with the packet unless ARP<sup>[Plum82]</sup> is in use. The implementation is thus able to deal only with Ethernet<sup>5</sup> networks and point-to-point links, for which the concept of destination address is not relevant. This is obviously easy to fix.

System V versions typically use the *Data Link Provider Interface* (DLPI)<sup>[McGr89]</sup> protocol between IP and the device driver. The `Pnet` server must implement its half of this protocol, a non-trivial matter. DLPI does provide for the destination address to be passed along. Unfortunately, it also introduces another complication at shutdown: the protocol requires that a link be unbound at connection tear-down, via a `DL_UNBIND_REQ` message and acknowledgement. This is not difficult for a resident device driver, but is problematic when the “device” is a pipe. Shutdown can occur when a server program has exited; there is obviously no way for the server to receive or send any more messages.

We worked with a pre-release version of the System V Release 4 streams TCP/IP, based on the Lachman/Convergent code; for it, some of these concerns were minimized. For example, although the drivers do acknowledge IP’s `DL_UNBIND_REQ` message, the acknowledgement is silently ignored; thus, its absence is not missed. Similarly, while some implementation-specific details — for example, associating the stream with a statistics structure, and actually keeping

5. Ethernet is a registered trademark of Xerox Corporation.

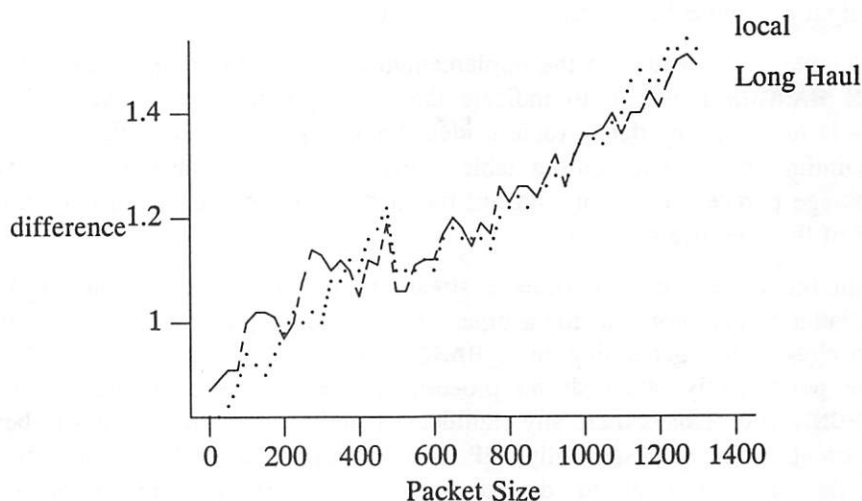


Figure 6. Time Differences

counts in that structure — are messy, the existing drivers in our version ignored them, so we ignored them as well.

Given that, we must implement the following aspects of the protocol:

- Respond to `DL_BIND_REQ` with a `DL_BIND_ACK` message. Since both of these messages are transmitted as `M_PROTO` streams messages, they could be sent and received easily enough via `putmsg()` and `getmsg()`.
- Respond to a `DL_INFO_REQ` message with a `DL_INFO_ACK` message. Again, this requires no kernel code.
- Accept and send data via `DL_UNITDATA_REQ` and `DL_UNITDATA_IND`.
- Accept a few `ioctl()` calls. This version of IP requires that the socket `ioctl()` calls, notably `SIOCSIFFLAGS`, `SIOCSIFADDR`, and `SIOCSIFNAME` (to set the interface structure name) be fielded by the driver (or a convergence module), and an `M_IOCACK` message sent back upstream. This one is more difficult, since there is no way to process `M_IOCTL` messages at the stream head, or to generate responses.

We could have implemented this via a special-purpose module. Indeed, if a module were needed anyway, to handle `DL_UNBIND_REQ`, we would probably have opted for that solution. Given that everything else could be handled at user level, though, we provided a general alternative, the `mesg/rmesg` module pair used in 9th Edition systems. These modules encapsulate all stream messages, regardless of type, as an `M_DATA` message preceeded by an `M_PROTO` header. In the reverse direction, a user-generated header is examined to produce an

arbitrary-type message from the data portion written via `putmsg()`.<sup>6</sup> A consequence of this is that even the DLPI messages are encapsulated this way; thus, the user process is slightly more complex than might otherwise be the case.

A few minor changes were needed to the implementation of IP. Most important, IP needs to recognize the `M_HANGUP` message, to indicate that the pipe has been closed. The proper response to this is to delete the data structure identifying a stream, and to delete any routing table entries pointing to it. The routing table adjustments should also be made when an `I_UNLINK` message is received for any stream; the lack of such could be considered a bug in IP regardless of the presence of `Pnet`.

Finally, although the current code permits a stream to be attached via either `I_LINK` or `I_PLINK`, the latter is inappropriate for a pipe. If the owning process dies, the user end of the pipe will be closed, thus generating an `M_HANGUP` and disabling the stream. The IP end, though, will be permanently attached; no process is likely to come along and issue the appropriate `I_PUNLINK`. Nor is there any significant benefit to the user process in being able to do a persistent link. Consequently, IP should reject `I_PLINK` calls for pipes. Unfortunately, that is not easy to do; the check is very implementation-dependent. Consequently, we have omitted it in this prototype.

## 7. CONCLUSIONS

We have demonstrated how one simple piece of code can be used to create a variety of powerful mechanisms. Given comparatively minor changes to the stream versions of IP, it was simpler yet. We have implemented some of the applications described above; work on others is in progress, notably `Greger`.

## REFERENCES

- [BFE] "Blacker Front End Interface Control Document," pp. 1-25-1-40 in *DDN Protocol Handbook*, ed. E.J. Feinler, O.J. Jacobsen, M.K. Stahl, and C.A. Ward.
- [Bell89] S.M. Bellovin, "Security Problems in the TCP/IP Protocol Suite," *Computer Communications Review* 19(2), pp. 32-48 (April, 1989).
- [Brad89] R.T. Braden, ed., "Requirements for Internet hosts - communication layers.," RFC 1122 (October 1989).
- [Come88] D. Comer, *Internetworking with TCP/IP : Principles, Protocols, and Architecture*, Prentice-Hall, Inc. (1988).
- [Denn81] D.E. Denning and G.M. Sacco, "Timestamps in Key Distribution Protocols," *Communications of the ACM* 24(8), pp. 533-536, ACM (August 1981).

---

6. In practice, life is a bit more complex; `M_FLUSH` messages must be processed both in the kernel and sent to the user process. Furthermore, security considerations dictate that use of `mesg/zmesg` be restricted to the superuser.

- [Estr89] D. Estrin, J.C. Mogul, and G. Tsudik, "Visa Protocols for Controlling Inter-Organization Datagram Flow," *IEEE Journal on Selected Areas in Communications* 7(4), pp. 486-498, (Special Issue on Secure Communications) (May 1989).
- [Fein85] E.J. Feinler, O.J. Jacobsen, M.K. Stahl, and C.A. Ward, *DDN Protocol Handbook*, DDN Network Information Center, SRI International (1985).
- [Jaco88] V. Jacobson, "Congestion Avoidance and Control," pp. 314-329 in *Proceedings of SIGCOMM '88* (August 1988).
- [Karn87] P. Karn and C. Partridge, "Improving Round-Trip Estimates in Reliable Transport Protocols," pp. 2-7 in *Proceedings of SIGCOMM '87* (August 1987).
- [Lanz89] L. Lanzillo and C. Partridge, "Implementation of Dial-Up IP for UNIX Systems," in *Proc. Winter USENIX Conference*, San Diego, California (January, 1989).
- [McGr89] G.J. McGrath, "DPLI Interface Specifications," AT&T White Paper (February 1989).
- [Mogu89] J. Mogul, "Simple and Flexible Datagram Access Controls for UNIX-based Gateways," in *Proc. Summer USENIX Conference*, Baltimore, Maryland (June, 1989).
- [Morr85] R.T. Morris, "A Weakness in the 4.2BSD UNIX TCP/IP Software," Computing Science Technical Report No. 117, AT&T Bell Laboratories, Murray Hill, New Jersey (February 1985).
- [Mund87] G.R. Mundy and R.W. Shirey, "Defense Data Network Security Architecture," in *Proc. MILCOM '87*, IEEE, Washington, D.C. (1987).
- [Need78] R.M. Needham and M. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM* 21(12), pp. 993-999, ACM (December, 1978).
- [Need87] R.M. Needham and M. Schroeder, "Authentication Revisited," *Operating Systems Review* 21(1), p. 7 (January 1987).
- [Nowi89] B. Nowicki, "Transport Issues in the Network File System," *Computer Communications Review* 19(2), pp. 16-20 (April, 1989).
- [Plum82] D.C. Plummer, "Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware," RFC 826 (November 1982).
- [Ritc84] D.M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal* 63(8, part 2), pp. 1897-1910 (October 1984).
- [Romk88] J.L. Romkey, "Nonstandard for transmission of IP datagrams over serial lines: SLIP," RFC 1055 (June 1988).
- [SP3] SDNS Protocol and Signalling Working Group, SP3 Sub-Group, "SDNS Secure Data Networking System Security Protocol 3 (SP3)," SDN.301 (July 12, 1988).
- [Ste88] J. Steiner, C. Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Proc. Winter USENIX Conference*, Dallas (1988).





**Steven M. Bellovin**  
AT&T Bell Laboratories

Steven M. Bellovin received a B.A. degree from Columbia University, and an M.S. and Ph.D. in Computer Science from the University of North Carolina at Chapel Hill. While a graduate student, he wrote the original version of pathalias and helped create netnews. He has been at AT&T Bell Laboratories since 1982, where he does research in networks, security, and why the two don't get along.

# Serpent: A User Interface Environment

Len Bass, Software Engineering Institute, Carnegie Mellon University  
Brian Clapper, Naval Air Development Center  
Erik Hardy, Software Engineering Institute, Carnegie Mellon University  
Rick Kazman, Department of Philosophy, Carnegie Mellon University  
Robert Seacord, Software Engineering Institute, Carnegie Mellon University

*This work was sponsored by the Department of Defense.*

## Abstract

Prototyping has been shown to simplify system specification and implementation, especially in the area of user interfaces. Previous prototyping approaches do not allow for the evolution of the prototype into a production system and do not support maintenance after a system is put into use. A set of goals for a modern user interface environment is presented. Serpent, a prototype implementation which achieves these goals, is presented and is shown to have several advantages over other approaches to user interface development.

## Introduction

The advent of the modern graphics-oriented workstation has placed increasing emphasis on user interface design. End users are increasingly more adamant that software should be both functional and easy to use. In response, both software and hardware vendors are paying more attention to the user interfaces that accompany their products. The importance of this issue can be measured by the intensity of the current industry struggle to establish user interface standards, such as the IEEE P1201 committee efforts.

Despite this awareness, a number of problems still exist. The user interface for a large system accounts for a large portion of the life cycle costs; in some systems, the user interface development and maintenance cost exceeds 50% of the total software cost [1]. Further, the process of user interface development is labor-intensive. Current user interface development tools and methods inadequately address this problem. In particular, while more and more vendors are providing user interface toolkits and graphics packages, these packages typically require extensive and specific knowledge of a particular toolkit or user interface library. These packages also require one to use conventional, procedural languages such as C and Ada. These languages are not particularly well-suited to user interface specification and implementation, so the user is coerced into worrying about low-level syntactic issues.

## The Case for Evolutionary Development

One method for increasing customer satisfaction with the user interface has been to develop the user interface prior to developing the final product. This development of a prototype has been shown to be effective [2]. A prototype is also used for conveying specifications since written specifications are inadequate for conveying the look and feel of the interface. Unfortunately, the complexity of the user interface packages currently available hinders efforts to evolve the prototypes into usable, maintainable, and enhanceable production systems. Although rapid prototyping is invaluable in the design process, complicated prototypes require a significant investment in time and effort. By forcing developers to discard the prototype after using it as a specification, substantial effort is required to implement the specified user interface. Consequently, a development approach that allows for the evolution of a user interface prototype into one that can be used in a fielded system and one that can be maintained and modified promises great savings to the total cost of a system.

## Previous User Interface Approaches

Early user interface development efforts were marked by intense coding in traditional programming languages of both the user interface and the underlying application. This approach is cumbersome and error-prone, due to the low-level semantics of these languages. Using this process, changes to the user interface specification may force major changes in the application program. Even though the prototype may have only addressed some limited portion of the overall requirements, there is a natural tendency to use it as a basis for the deliverable product.

Later, specialized prototyping languages were developed, employing specific shorthand notations to generate corresponding function invocations [6]. These languages are usually fairly arcane, not unlike RPG and its successors, in that the user interface designer must be intimately familiar not only with the language, but also with the built-in functions. One of the big drawbacks to this approach is that once the prototype is done new code must be developed to implement the deliverable user interface, using the prototype as its requirements. There is no smooth transition from prototype to product.

With the advent of fourth generation languages and the increased use of computers for management information systems came the concept of rapid prototyping [4]. This approach is marked by the application of database concepts to software development; changing a value in the database causes a resultant change in the presentation. One major advantage over other approaches is that, for each end-user invokable function, there is a corresponding program-callable routine. Once the user interface is specified, the appropriate calls can be made by the application program. However, if the user interface changes, the application program must be changed.

The explosion in workstation capabilities in the last few years has sparked many new ideas about how to use these capabilities for user interface development [9, 10, 8, 3, 5, 7], leading to a multitude of tools and environments, such as Prototyper, XVT, UIL, Granite, Autocode, and MIKE. However, each is marked by the use of a

specific language and/or interactive tools tailored to the capabilities of a particular platform and/or to the specific user interface technology supported. Application support in these packages usually takes the form of a fixed set of functions that can be invoked as necessary by the application or a set of functions that are dynamically generated by the prototyping tool to implement the user interface. Again, if the user interface changes, the application must be changed to invoke the new functions.

Finally, user interface technology is evolving rapidly. Today's leading edge data presentation theory becomes tomorrow's commonplace toolkit, giving way to some previously unimagined technology. None of the above approaches adequately provides for the effective integration and use of new technologies.

### **Goals of a Modern User Interface Environment**

In 1987 the Software Engineering Institute started the User Interface Project to address perceived problems in user interface development and to assist the transition of user interface design and development technology into practice. Out of this effort arose a set of goals for the next generation of user interface environments.

1. In any computer system, there should be a true separation of concerns between the application and the user interface. This is simply the concept of modularity; the application should not try to perform the functions of a user interface, and vice versa. One should be able to develop the application independently of the user interface, in a language appropriate to the semantics of the application. Similarly, user interface development should be independent of the application.
2. There should be a single, consistent application program interface (API), and the API should make no preconceptions about the nature of the user interface. This allows the application to remain unchanged, even though the user interface may change.
3. The user interface specification, design, and implementation should be simple and straightforward; prototyping should be fairly easy using the mechanisms provided by the environment. Non-programmers should be able to perform these activities with a minimum of training. The mechanisms used to perform these activities should not have to change, even though the user interface style or underlying user interface technology may change.
4. It should be possible to prototype the functionality of a system without an application. The user interface support mechanisms should be sufficiently rich to support reasonably sophisticated prototypes. As the prototype matures, facilities should be provided to add an application, in pieces or all at once, thus providing evolutionary development. This also allows a completed prototype to be included as part of the deliverable system.
5. Existing systems should be able to take advantage of new technologies as they become available, without affecting the application portion of the system. The mechanisms for incorporating new technologies should be relatively simple.
6. Performance, when the environment is used strictly as a prototyping vehicle, should be reasonable, although special performance considerations would have to be made when used in production.



## Serpent

Starting with the above goals, the User Interface Project developed a user interface environment known as Serpent. Serpent is a user interface management system (UIMS), using the standard Seeheim model [11], that supports the development and execution of the user interface of a software system. Serpent supports incremental development of the user interface from the prototyping phase through production and maintenance. Serpent encourages the separation of concerns between the user interface and the functional portions of an application. Serpent is easily extended to support multiple input/output technologies.

## Architecture

Figure 1 shows the overall architecture of Serpent. The architecture is intended to encourage the proper separation of functionality between the application and the user interface portions of a software system. The three different layers of the architecture provide differing levels of control over user input and system output. The *presentation layer* is responsible for layout and device issues. The *dialogue layer* specifies the presentation of application information and user interactions. The *application layer* provides the actual system functionality.

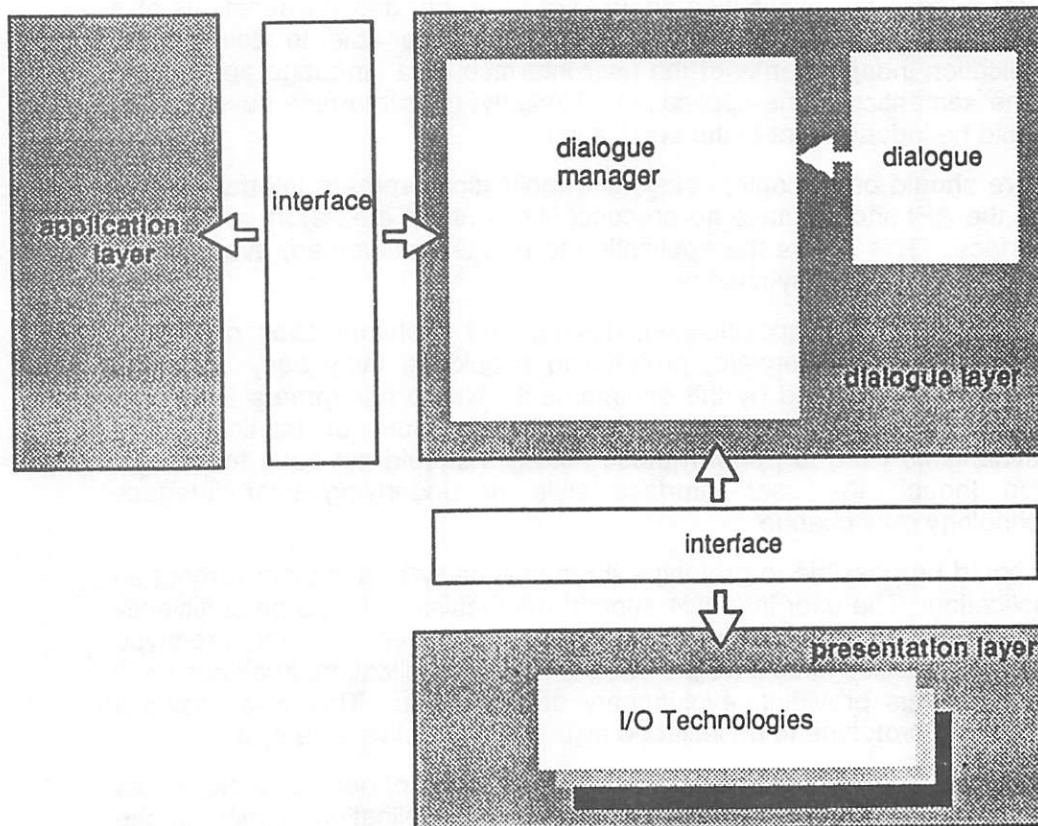


Figure 1: Serpent Architecture



The presentation layer controls the end-user interactions and generates low-level feedback. This layer consists of various input/output technologies that have been incorporated into Serpent. A standard interface has been defined which simplifies adding new technologies. Each technology defines a collection of *interaction objects* visible to the end user.

The dialogue layer specifies the user interface and provides the mapping between the presentation and application layers. The dialogue layer determines which information is currently available to the end user and specifies the form that the presentation will take as previously defined by the dialogue writer (individual responsible for creating the user interface specification, or *dialogue*). The dialogue layer acts like a traffic manager for communication between application and technologies. The presentation level manages the presentation; the dialogue layer tells the presentation what to do. For example, the presentation layer manages a button that the end user can select; the dialogue layer informs the presentation layer of the position and contents of the button and will act when the button is selected.

The application layer performs those functions that are specific to the application. Since the other two layers are designed to take care of all the user interface details, the application can be written to be presentation-independent; there should be no dependency in the application on a specific technology.

The data that is passed between different layers is known as *shared data*. Data passed between an application and the dialogue layer is referred to as application shared data, while data passed between a technology and the dialogue layer is called technology shared data. A *shared data definition* provides the format of the data.

## Slang

In Serpent, user interface dialogues are specified in a special-purpose language called Slang. Slang provides a mechanism for defining the presentation of information to, as well as interactions with, the end user without enforcing specific user interface policies. A Slang program defines and enumerates a collection of interaction objects and actions available to the end user.

The interaction objects available to the dialogue writer are defined by the input/output technology. Each technology defines a set of primitive objects that may be used in a dialogue. Each object has a collection of *attributes* that define its presentation and a collection of *methods* that determine how the end user can interact with that object.

Slang also provides variables for intermediate storage and manipulation, along with a full complement of primitive arithmetic operations. The specification of an attribute or variable can take one of three forms:

1. A constant value
2. The identifier of another attribute or variable within the current lexical scope
3. A *code snippet*, which is a fragment of procedural code that calculates the value that the attribute or variable is to take. A code snippet can also refer to other attributes and variables within the current scope

By contrast, a method usually takes the form of a code snippet that is executed once each time the method is executed.

In Slang, dependencies between items are automatically enforced. That is, suppose variable *V* depends on the value of some object attribute *A*. If *A* changes (perhaps due to some end user action), the value of *V* is re-evaluated automatically. This important and powerful feature allows the dialogue writer to build complex, interdependent interaction objects simply by specifying a series of dependencies.

Slang also allows a dialogue writer to group arbitrary objects into logical collections called *view controllers* that may be created or destroyed as a unit. Specifying a view controller in Slang defines a *view controller template*. Each template has a *creation condition* that defines when an *instance* of the template should come into existence. The existence of a view controller instance and its child objects can be controlled by the values of Slang variables or by the creation or destruction of application data. When a view controller's creation condition is no longer valid, the view controller and its associated objects are destroyed. Multiple instances of a template may exist at any time. A view controller serves two main purposes.

1. It maps specific application data onto display objects with which the end user can interact.
2. It controls the existence of a series of related objects.

## Application Program Interface (API)

From the application developer's perspective, Serpent behaves like a database management system. Shared data is manipulated in the database by the application, the presentation layer (usually in response to end-user actions), or the dialogue layer (in response to actions within the dialogue).

The application can add, modify, or delete shared data. Information provided to Serpent by the application is available for presentation to the end user. The application has no interface to the presentation layer and therefore cannot affect how data is presented to the user. When end user actions cause the dialogue to change the application shared database, the application is automatically informed. In this sense, the application views Serpent as an active database manager.

## Saddle

The type and structure of data that is maintained in the shared database is specified in a shared data definition file, defined in a language called Saddle. This data definition corresponds to the database concept of *schema*. A shared data definition file is created once for each application and once for each technology that is integrated into Serpent. A shared data definition is required before an application can use Serpent.

The shared data definition file is processed to produce a language-specific description of shared data. Processors currently exist for Ada and C. If the application is written in C, the processor will generate structure definitions that can be included into the application program. If the application is written in Ada, the processor will generate package specifications.

## Technology Integration

Given that Serpent manipulates objects, the technologies that are most effortlessly integrated are those that are object-oriented. The successful integration of object-oriented graphics systems and their associated toolkits has been a major proof of Serpent's abilities to separate presentation concerns from application concerns.

The process of integrating a technology into Serpent is conceptually simple. It can be logically divided into three parts.

1. The objects with which the end user will interact must be determined, along with their behavior.
2. The objects must be defined in Serpent through the use of Saddle.
3. A driver must be written to allow the technology to communicate with the dialogue manager, through Serpent's shared database facility.

If a technology already has an object orientation, then the first and third integration steps are usually straightforward. If it does not, then a set of objects and their attributes must be built on top of the technology to conform to the Serpent model, involving potentially difficult mapping issues.

Technology integration presents other practical difficulties. The integrator has to decide how much of the underlying technology to expose to a dialogue writer, whether to change any of the default behavior of the system, and whether to make the system more robust by, for instance, performing error checking that the technology does not handle.

## The Serpent Development Process

Slang was designed explicitly for user interface specification. A Slang dialogue writer is not burdened with the technical and procedural details necessary to manipulate specific interaction objects; those details are hidden in the presentation layer. The dialogue writer merely specifies the objects that make up the user interface and indicates how they relate to one another and to the end user; the Serpent runtime system manages the interaction objects. The dialogue writer needs to be familiar with the characteristics of various objects, such as knowing that an Athena Widget Set label widget appears as a rectangle on the screen; however, the writer does not need to know how to tell the X Toolkit library how to display such a widget.

Slang dialogues can be executed without the benefit of an application, allowing one to build, test and refine a prototype before designing and implementing the rest of a system. Often, however, a prototype requires the existence of some application functionality, if only to initialize display values. Slang's rich set of primitive operations allow the user interface designer to "mock up" application operations in the prototype dialogue. Once the prototype has been refined, the simulated application behavior can be removed from the dialogue.

## A Simple Example

Perhaps the best way to illustrate the simplicity of prototyping with Slang is by example. Figure 2 shows the screen display for a counter demonstration, using the X Toolkit Athena Widget Set. The box labeled "PRESS" is a command widget that can be selected by the end user via a mouse. The box above the command widget is a label widget containing the current value of the counter. When the end user selects the button labeled "PRESS", the value in the label widget is incremented by 1.

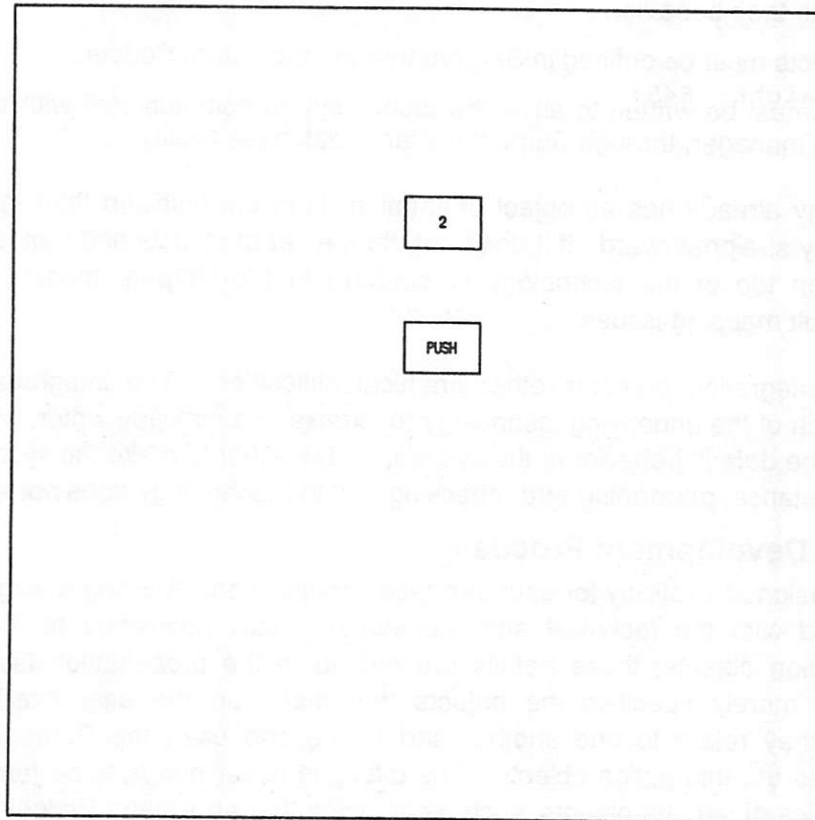


Figure 2: Counter: A Simple Example

In Slang this example is written as follows:

```
VARIABLES:
    counter : 0;

OBJECTS:
    /*
       width, height, vert_distance, and horiz_distance
       are all specified in pixels
    */
    background: form_widget
    {
        ATTRIBUTES:
            width: 640;
            height: 645;
    }

    display: label_widget
    {
        ATTRIBUTES:

            parent:      background;
            width:       60;
            height:      40;
            vert_distance: 150; /*from upper left of parent*/
            horiz_distance: 310; /*from upper left of parent*/
            label_text:  counter;
    }

    push_button: command_widget
    {
        ATTRIBUTES:

            parent:      background;
            width:       60;
            height:      40;
            vert_distance: 250; /*from upper left of parent*/
            horiz_distance: 310; /*from upper left of parent*/
            label_text:  "PRESS";

        METHODS:
            notify:
            {
                counter := counter + 1;
            }
    }
}
```

The **background** object provides a form on which to locate the other objects. The **display** object defines the label widget containing the current value of the counter; note that the **label\_text** field, which controls what is actually displayed in the form widget, is dependent on the value of the global variable **counter**. When the value of the variable changes, all items that depend on it are re-evaluated. Put more simply, if **counter** changes, the text displayed in the **display** object will change automatically.



The **push\_button** object defines the command widget that the end user will select in order to increment the value displayed on the screen. When the end user selects the button, the presentation layer captures the event and communicates it to the dialogue via a **notify** method, causing the associated code snippet to be executed. In this case the **counter** variable is incremented which, in turn, causes the label in the **display** object to be changed.

Dependencies and type conversions are managed automatically by the Serpent runtime system, allowing the dialogue writer to focus on user interface issues, rather than syntactic details. For example, the **counter** variable is an integer; the **label\_text** attribute of the **display** object is a string. Slang converts the **counter** value to a string before assigning it to the **label\_text** attribute; the dialogue writer merely needs to specify the dependence between the variable and the attribute. Further, the attributes for every interaction object take reasonable defaults, so the dialogue writer does not need to specify a value for every possible characteristic of an object.

In short, Slang is designed to minimize the amount of information the dialogue writer needs to specify in order to manipulate interaction objects.

### A More Complex Example

A more complicated example can serve as a better case study of how one builds a prototype of a user interface with Slang. The application system described in this section was actually built by a member of the Serpent development team. The goal was to produce a version of the once-popular television game *Concentration*. Figure 3 shows the final product during a game.

The game was developed in several phases:

1. Build a prototype of the playing board.
2. Add the logic necessary to play the game. Use a constant set of data for the prizes and locations during this phase.
3. Polish the prototype until it was satisfactory
4. Add an application to select a random set of prizes and locations and ship them over to the dialogue.

Phases 1 and 2 were completed in approximately an hour and were programmed entirely in Slang. The game-playing logic could as easily have been relegated to the application; however, the game-playing logic is conceptually simple and including it in the dialogue allowed the prototype to be built more quickly. Once the prototype was running, it was refined until the developer was happy with the way it appeared.

Finally the application, the simplest part of the game, was written. It merely selects the prizes and places them into the shared database. The dialogue then populates the game board with those prizes and manages the game. Implementing the application and making the necessary minor adjustments to the dialogue involved approximately two hours of work.

Concentration				
QUIT	No match. Press OK to continue.			OK
1			4	
6	7	8	9	10
11	12	13	An Ancient VAX	15
16	Beat-up Chevy	18	19	20
	22	23	24	
26		28	29	30
Player 1 Score: 2		Player 2 is up		Player 2 Score: 1

Figure 3: Concentration: A More Complex Example

## Status

The initial implementation of Serpent was done under ULTRIX 2.2 on DEC microVAX II and III workstations. Serpent was also easily ported to run under SUNOS 3.5 or higher on SUN2 and SUN3 workstations. We expect porting to similar UNIX platforms to be relatively straightforward. Other Serpent users are currently porting Serpent to the Sun 386i and DECStation 3100 platforms.

Applications can be written in either C or Ada, and simple mechanisms exist to extend Serpent to support other high level languages. Serpent was implemented predominantly in C, with additional support software written as shell scripts.

Currently, three different interfaces to the X Window System have been written for Serpent: one implements a subset of the Athena widget set, another uses the Athena

widgets supplemented by specially developed widgets for drawing, and a third implements a subset of the Motif widget set. In addition, Lockheed's Softcopy Map Display System has been integrated. Other integration efforts currently planned include an integrated video/graphics workstation, a video mapping system, an experimental gesturing device, and XView--Sun's implementation of the Open Look User Interface.

An interactive graphical editor is under development that hides most of the details of the user interface specification, and an interactive graphical debugger has been designed.

Serpent is available from the Software Engineering Institute through anonymous ftp and is also contained in the X11R4 contrib release at MIT.

## Conclusions

As a result of our experiences in developing user interfaces with Serpent, we have concluded that Serpent offers the following advantages over other user interface development approaches:

1. The active database model for applications allows the true separation of application issues from user interface issues, thus ensuring modularity. Application writers are also free from the syntactic drudgery inherent in programming large, complex input/output technologies.
2. The constraint mechanisms implemented via automatic dependency updates ensure that all participants (application, dialogue manager, and technology) are synchronized in terms of the state of the system.
3. Serpent's language-independent interface definition and inter-process communication mechanisms help in achieving modularity. Application developers are not constrained to work in a single language.
4. Serpent's technology integration support reduces the integration process to a series of concise, well-understood steps. Once a particular technology is integrated, its objects are available for use in any dialogue.
5. Due to Serpent's inherent separation of concerns, system developers can experiment with different user interface styles, and even different technologies, without changing either the application code or the API. This also provides for the injection of new technologies and user interface paradigms into an existing system, while minimizing the system portions which are impacted.

Serpent has achieved the goals of a modern user interface environment set forth earlier. The user interface specification mechanisms are simple and direct; changes in the user interface are made easily, without changing the application. The application program interface is simple and easy to use and enforces a true separation between the application and the user interface portions of the system. Prototyping is accomplished rapidly, with reasonable provision for application functionality simulation. Serpent's technology integration mechanisms allow a new technology to be incorporated into Serpent easily without affecting the application. Finally, Serpent is itself a prototype, implementing the goals listed above. Even so, performance is quite reasonable, and we are continually making improvements, although we would not yet recommend it for time-critical production environments.

## References

- [1] Boehm, Barry W.  
A Spiral Model of Software Development and Enhancement.  
*Computer* 21(5), May, 1988.
- [2] Boehm, Barry W.  
Improving Software Productivity.  
*Computer* 20(9), September, 1987.
- [3] Colborn, Kate.  
OSF Determines User Interface; Choices Could Affect the Development of  
Applications Software.  
*EDN*, December, 1988.
- [4] Fisher, Gary E.  
*Application Software Prototyping and Fourth Generation Languages*.  
Technical Report, National Bureau of Standards, May, 1987.
- [5] Foley, James, et al.  
Defining Interfaces at a High Level of Abstraction.  
*IEEE Software*, January, 1989.
- [6] Hanner, Mark Allen.  
Gambling on Window Systems.  
*UNIX Review*, December, 1988.
- [7] Kasik, David J., et al.  
Reflections on Using a UIMS for Complex Applications.  
*IEEE Software*, January, 1989.
- [8] Kolodziej, Stan.  
User Interface Management Systems.  
*Computerworld*, July 8, 1987.
- [9] Myers, Brad A.  
*Tools for Creating User Interfaces: an Introduction and Survey*.  
Technical Report CMU-CS-88-107, Carnegie Mellon University, 1988.
- [10] Myers, Brad A.  
*The Garnet User Interface Development Environment: a Proposal*.  
Technical Report CMU-CS-88-153, Carnegie Mellon University, 1988.
- [11] Pfaff, G. (Ed.).  
*User Interface Management Systems*.  
Springer-Verlag, Berlin, 1985.



**Len Bass**  
CMU

Len Bass (ljb@sei.cmu.edu) is currently the head of the User Interface Project at the Software Engineering Institute. He has been at the SEI since 1986 and prior to joining the SEI, was on the faculty of the University of Rhode Island. Before coming to his current interest in user interfaces, he has working in the areas of theory of computation, operating systems, and data bases.



**Rick Kazman**  
CMU

Rick Kazman (rnk@sei.cmu.edu) is a Ph.D. candidate in Computational Linguistics at Carnegie Mellon University and member of the User Interface Project at the Software Engineering Institute. He has been working at the SEI since 1988 and has been attending CMU since 1986. His current research interests are: graphical user interfaces; the modeling of human language acquisition; structured text files; and computational lexicography.



**Brian Clapper**  
Naval Air Develop. Center

Brian Clapper (clapper@nadc.arpa) is a computer scientist employed by the Naval Air Development Center in Warmister, PA. Since February, 1989, he has been assigned to the Software Engineering Institute where he has been working on the User Interface Project.



**Robert Seacord**  
CMU

Robert Seacord (rcs@sei.cmu.edu) is a Member of the Technical Staff in the User Interface Project at the Software Engineering Institute (SEI). Before joining the SEI, he was employed by the IBM Corporation where he worked in the areas of software engineering, processor development, and communications. Seacord earned a bachelor of science degree in computer science from Rensselaer Polytechnic Institute.



**Erik Hardy**  
CMU

Erik J. Hardy (ejh@sei.cmu.edu) is currently on the staff of the Software Engineering Institute, where he works on the User Interface Project. Prior to his interest in user interface prototyping and design, he taught Mathematics at Carnegie Mellon University, and he has written several Mathematics textbooks.



# Parallel Object-Oriented UIMS with Macro and Micro Stubs

*Masami Hagiya, Kyoto University*  
hagiya@kurims.kyoto-u.ac.jp

*Kouji Ohtani, ASTEC Inc.*  
sho@astec.co.jp

## Abstract

This paper describes a user-interface management system (UIMS) that is based on a parallel object-oriented model and supports a stub generator for connecting the user-interface part of an application running inside the UIMS and the application body implementing application-specific functionalities. The parallel object-oriented model is implemented by a virtual machine in the window server, which allows entire user-interface code to run inside the server. The UIMS supports two kinds of stub: macro and micro. With a macro stub, an entire UNIX process behaves as a parallel object in the parallel object-oriented model of the UIMS, while with a micro stub, each object implemented inside a UNIX process corresponds to a parallel object in the model.

## 1. Introduction

Parallelism inherent in graphical user-interface is one of the key issues in studying the methodology for developing interactive applications on window systems. Conceptually, there must exist an independent control thread for each activity in graphical user-interface. For example, for each software button, there must be a control thread which reads an input event corresponding to a mouse button click on the software button and performs the action specified for the software button. Similarly, each pop-up menu must have a control thread of its own.

Some window systems, such as the previous version of GMW — Give me More Windows — [7], support an object-oriented programming language with light-weight processes, where each activity in graphical user-interface is implemented by a separate light-weight process. In NeWS [11], its extended PostScript language supports light-weight processes and PostScript is neatly used as an object-oriented programming language. Smalltalk-80 [4] is also a typical example, in which a process is allocated to the controller of each triple in the MVC model.

In non-parallel object-oriented programming languages, there are three basic abstractions: object, message and process, where a process is an active agent which runs methods of objects it controls. In those languages, sending a message is nothing more than calling an overloaded function, because an object is not a unit of exclusive access, i.e., messages sent to an object are not queued. Moreover, in window systems implemented in those languages, events from external devices or from windows are usually represented as objects, not as messages, and a process reads events by explicitly executing input statements.

Parallel (or Concurrent) object-oriented programming [13] is the programming paradigm that merges the two abstractions: object and process. In parallel object-oriented programming languages, all the objects are independent active agents whose execution is initiated

by messages sent to them. Messages to an object are queued, because an object can process a message at a time.

In building graphical user-interface by parallel object-oriented programming, programmers do not have to explicitly spawn a process for each possible activity in user-interface, because each object is already an active agent from its creation. An event can be naturally represented as a message and an object is a natural extension of an event-handler. This means that the so-called *external control (dialogue dominant) model* of UIMS [6], which has several advantages over the *internal control (computation dominant) model*, can be realized by regarding the invocation of a function of an application as a message sent to the application.

The idea of implementing window systems under a parallel (or concurrent) object-oriented model also appears in [1], but the idea was not examined in detail in an actual implementation.

### 1.1. Overview of GMW Window System Version 3

GMW (Give me More Windows) is a general-purpose multi-window system being developed by Kyoto University, OMRON Tateisi CO. and ASTEC Inc. for workstations running under the UNIX operating system. The previous version of GMW is reported in [7], where the imaging model and the virtual-machine-server architecture of GMW are fully described. This paper describes the latest version (Version 3) of the system, which supports an environment for developing and implementing the user-interface part of an application completely inside the window system (inside the window server, precisely speaking), and thus should be classified not as a window system but rather as a user-interface management system (UIMS).

The characteristic features of this version of GMW are:

1. the virtual machine and the high-level language based on a parallel object-oriented model, which are exclusively devoted to implementing graphical user-interface,
2. the stub generator for connecting the user-interface part of an application and the application body, and
3. the direct manipulation style environment for developing the presentation component of user-interface.

### Parallel Object-oriented Virtual Machine

As is described in [7], GMW supports a virtual machine for realizing the extensibility of the window system. The virtual machine of the previous version, like Smalltalk-80 or NeWS, supported light-weight processes for handling the concurrency inherent in graphical user-interface. While developing the new version, however, we reached the above conclusion, and designed a parallel object-oriented programming language, called G, and redesigned the virtual machine as an interpreter for parallel objects in the language.

The most serious demerit of the approach with a virtual machine is the inefficiency of interpreting the virtual code. However, as many user-interface definition languages (UIDL) are interpreted, it is widely considered to be vital, sometimes more important than efficiency, for user-interface to be easily modifiable, and this justifies the approach with a virtual machine.

In fact, usual user-interface code runs with an enough speed in the current implementation of GMW.

### Application as Server

From the point of software architecture of a UIMS, how to implement the *application interface*, i.e., the interface between the user-interface part of an application and the application body is an important problem that determines the customizability of user-interface of an application. Our approach to this problem is to implement the interface with stubs for remote procedure calls [3].

Unlike the widely accepted server-client model for window systems, in which application programs are implemented as clients to the display server, in GMW, a server stub generated from a stub description is attached to an application program which works as a server to its user-interface part in the UIMS (see Figure 1.1). On the other hand, operations of the window system, including imaging primitives, are invoked from the application program through client stubs.

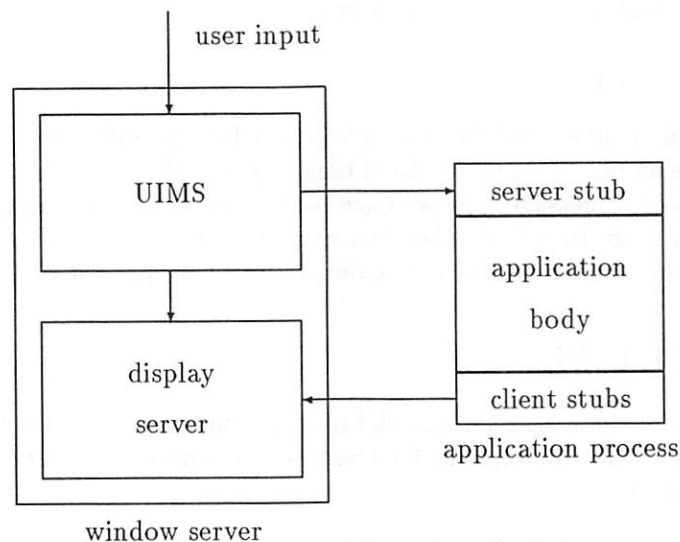


Figure 1.1. Basic Architecture of GMW

### Macro and Micro Stubs

GMW supports two kinds of stub: macro and micro. With a macro stub, an entire UNIX process, which implements the body of an application, behaves as a parallel object in the parallel object-oriented model of GMW. Each function of a UNIX process corresponds to a method of the parallel object. A UNIX process, after entering the message-handling loop, invokes its functions as it receives messages.

Micro stubs are supported for conventional non-parallel object-oriented programming languages like C++. With a micro stub, each object inside a UNIX process, implemented in an object-oriented programming language, behaves as a parallel object in the model. Micro stubs facilitate the implementation of applications like graphic editors, which handle an indefinite number of objects each of which has a separate user-interface part.

### Application Interface as IPC Interface

The importance of the identification of the application interface and the IPC (inter-process communication) interface can not be exaggerated too much. The user-interface part of an

application can be customized without recompiling or relinking the application body, even while an application body is running.

Another important result of the identification is that a single application is naturally and easily split into processes, each of which implements a specific functionality of the application and is placed at an appropriate location in a distributed environment. These processes are integrated through the user-interface part running inside the UIMS.

As distributed environments become more and more common, the very idea of running the entire user-interface part of an application at the location physically nearest to the user becomes more and more important. Although it is not required to merge a UIMS with a display server, we believe that since they are placed at the same physical location, the approach taken by GMW is promised to become a general way of organizing interactive softwares in distributed environments.

In order to prove the effectiveness of our approach, we are currently developing a window terminal server for GMW, where most of user-interface code runs inside the terminal.

## 1.2. Outline of the Paper

In this paper, we first give a brief overview of the UIMS by explaining its three layers in turn. We then give an introduction to the G language as well as our parallel object-oriented model based on colored messages. Experiences of the use of G in developing the UIMS and its applications are also described. We then explain the stub generator of the UIMS and how an application is constructed with it. Comparisons with related works are finally given.

## 2. Layers of UIMS

The UIMS consists of three layers of parallel objects which comprise the user-interface part of an application and the environment for creating and editing the user-interface part. The three layers are as follows:

1. operator interface — OIMO and keymaps,
2. presentation component — panels and tools,
3. dialogue control.

Each layer is implemented by a set of parallel objects and transforms lower-level messages from the previous layer into higher-level messages to the next layer as depicted in Figure 2.1. All the layers are implemented in the G language explained in the next section.

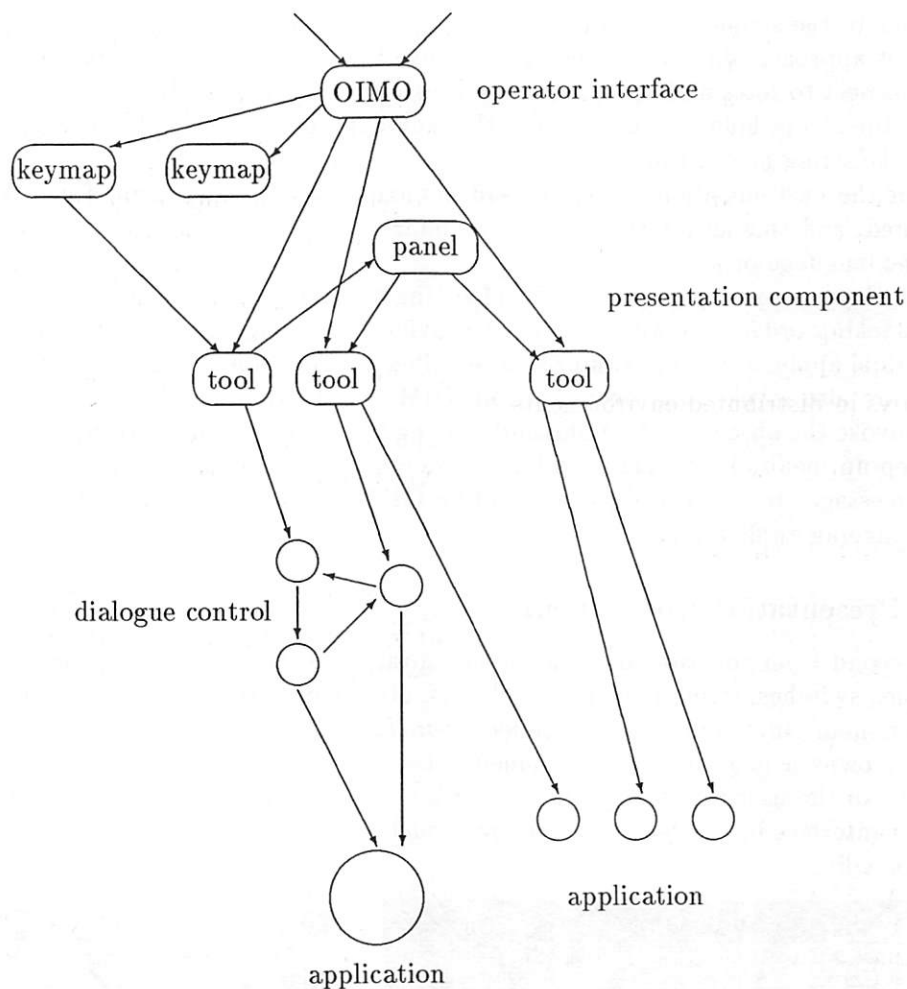


Figure 2.1. Layers of Parallel Objects

### 2.1. Operator Interface

Inputs from the user, such as mouse button clicks or key strokes, are sent to objects in the first layer. These inputs are in the form of a message, but are usually called *events*, because they roughly correspond to events in other window systems.

All the input events from the user are first sent to the object called *OIMO* (Operator Interface Message Organizer). *OIMO* analyzes each event based on the location of the mouse cursor at which the event was generated. Precisely speaking, *OIMO* checks if a receiver object is attached to the window of the event. If a receiver object is attached and satisfies certain conditions, *OIMO* sends an appropriate message to the receiver object. In the case of a mouse button event, the receiver is usually a *tool* object belonging to the second layer. For example, if a mouse button is pressed on the window of a software button, a **pressed** message is sent to the button tool controlling the software button. In the case of a key stroke event, the receiver is usually an object called a *keymap object*, which belongs to this layer. A keymap object analyzes input strings and sends appropriate messages to another object. For example, if the DELETE-key is typed while a mouse cursor is on a string holder tool, an input event corresponding to the DELETE-key is sent to the keymap object shared by string holder tools. This keymap object then sends an **erase\_char\_backward**



message to the string holder tool.

This approach allows the user to customize the operator interface in a uniform manner with respect to tools and applications. For example, since a single keymap object is shared by all the string holders, customizing this keymap object uniformly changes the behavior of all the string holder tools.

For the customization of OIMO and keymap objects, a number of control panels are prepared, and this allows the user to customize the operator interface without knowing a specific language or a file format.

The layer also includes objects for changing window configuration, such as position, size and stacking order of a window. They provide functionalities that are independent from individual applications and comprise a so-called *window manager* in other window systems. They are also invoked by messages from OIMO. For example, clicking the middle button may invoke the object for dragging and moving the window pointed to by the mouse cursor.

Pop-up menu objects can also be considered to belong to this layer. They are used to send messages to various objects including OIMO, window manager objects, and objects for launching applications.

## 2.2. Presentation Component

The second layer corresponds to a toolkit library in other window systems. Tools such as buttons, switches, string holders, scroll bars, etc. belong to this layer. A collection of tools that are manipulated in a unit are called a *panel* and the user interface part of an application usually owns a panel or a set of panels. Therefore, objects in this layer determines the outlook of the user-interface part and can be said to comprise the *presentation component* of user-interface in the sense of Seeheim model [5]. Figure 2.2 is an example of a panel for an icon editor.

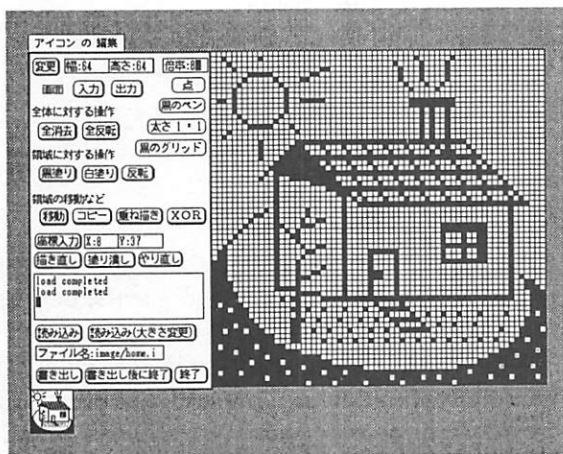


Figure 2.2. Icon Editor Panel

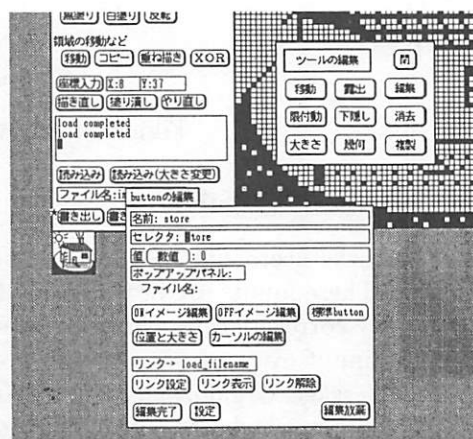


Figure 2.3. Editing Icon Editor Panel

A panel tool can send a message with appropriate arguments. For example, when the CR-key is typed on a string holder tool, it sends a message with the contents of the string as the message arguments. As is explained in Section 3.3, a message consists of a message selector and message arguments. The selector and arguments of messages sent by a tool are determined by the tool. The message receiver is determined by the panel to which the tool belongs and is usually set to the application body or a dialogue control object in the third layer.

The repertoire of panel tools supported by GMW is summarized in Table 2.1. The second column of the table tells when a message is sent from a tool. The third column tells message arguments. A tool belonging to a panel can have a link to another tool in the same panel. For example, a button tool can have a link to a string holder tool. In this case, when a button is pressed, it sends a message with the contents of the string holder as the message arguments.

tool type	when to send a message	message arguments
label	—	—
button	pressed and released	none, or contents of the linked tool
select switch	the state is changed	new state
alternate switch	pressed and released	new state
string holder	CR is typed	string contents
scroll bar	the bar is fixed	bar ratio
scroll menu	an item is selected	item number
canvas	a mouse button is clicked	button id and coordinates

Table 2.1. Panel Tools

A simple application can be built without the third layer, i.e., the message receiver of a panel is set to the application body and messages from panel tools are directly sent to it. For example, if the STORE button on Figure 2.2 is pressed, a `store` message is sent to the application body of the icon editor.

The UIMS supports a so-called *layout editor* for creating and editing panels, i.e., the presentation component of user-interface can be created and edited in a direct manipulation style environment without programming. Precisely speaking, each tool and panel object has a built-in facility for editing itself and dumping its state in the form of G code. Since these objects run in parallel with the application body, they can be edited even while the application body is running. Figure 2.3 shows a panel in Figure 2.2 being edited.

### 2.3. Dialogue Control

With only the second layer, however, one cannot create a user-interface part of an application that owns states and changes its behavior depending on its current state. The state transition occurs by messages from the presentation component and the application body. In Seeheim model, this component of user-interface is called a *dialogue control*.

Currently, the UIMS does not support a standard way to construct the dialogue control. One has to program it using the G language.

Messages from the dialogue control are sent to the application body.

## 3. Parallel Object-oriented Language G

G is a parallel object-oriented programming language designed to implement objects in graphical user-interface. G is based on the colored message model explained in Section 3.4 and supports the full spectrum of multiple-inheritance. G is also used to allow the user to interact with the window server as a command language is used to interact with the operating system. To satisfy these requirements, G was designed as an interpretive language with no types and no declarations.

### 3.1. Object

G is a *typeless* language in the sense that variables and expressions are untyped. Each variable holds a value of one word, which can be interpreted in various ways: signed integer, unsigned integer, character, object pointer, etc. An *object pointer* is the unique number given to each parallel object. An object is always referenced by its object pointer.

Although expressions are untyped, each parallel object of G has a type. A type of an object is called a *script*; scripts are also objects. There are some predefined scripts called *compiled scripts*. Objects whose type is a compiled script are used to define the basic constructs of G. Names and vectors are examples of such objects.

A *name* corresponds to a symbol in Lisp. A name has a string as its external representation, and only one name is created for one string. In G, a name is written with a period (.) before its string. The expression `.init` represents a name whose string is `init`, and is evaluated to the object pointer of the name. Names are used for various purposes. For example, a message consists of a name as its *message selector* and some arguments. A name can be associated with an object, i.e., an object can have a name. To denote such an object, an asterisk (\*) is placed before its name. The expression `*button` represents the object denoted by the name `.button`.

There are three kinds of *vector*: vector of words, vector of characters and vector of bytes. Elements of a vector are referenced with the conventional syntax that uses brackets ([ and ]). For example, if the variable `x` holds an object pointer of a vector, the expression `x[3]` references the fourth element of the vector.

### 3.2. Script

Each object has a script as its type, and the script determines the structure and behavior of the object. A script is also an object, so a message can be sent to a script object. A script has a set of *methods*, and a method consists of a *message selector* and a sequence of statements. When an object receives a message, the method whose selector matches with the selector of the message is invoked and its statements are executed.

The definition of a script begins with the name of the script and possibly a sequence of names of *super scripts*:

```
script button : tool
  on .init
    ...
  end on
  ...
end script
```

This script has the name `.button` and has one super script whose name is `.tool`. It has several methods; the first one has the selector `.init`. By this definition, the defined script object is given the name `.button`, and can be referenced by the expression `*button`.

### 3.3. Message Sending

Sending a message corresponds to calling a function in conventional programming languages. A message is sent by the syntax `receiver(selector)` or `receiver(selector, arguments)`. For example, `*button(.create)` sends a message with the selector `.create` to an object denoted

by the name `.button`, i.e., the script object defined in the previous example. Sending a `.create` message to a script object creates an *instance* of the script, i.e., an object whose type is the script, and returns its object pointer.

There are two ways of sending a message: *RPC* and *SEND*. *RPC* is synchronous message sending, whose sender blocks until it receives the reply (the return message) from the receiver of the message. For example, `x.do` waits for the reply from the object `x`. The return message of *RPC* can be assigned to a variable as in `r=x.do`.

*SEND* is asynchronous message sending, whose sender continues its execution after sending a message. *SEND* is denoted by an ampersand (`&`) as in `&x.do`.

### 3.4. Colored Message

The parallel object-oriented model of GMW is based on *colored messages*. Each message or object has a certain color. When an object is processing a message, it has the same color as that of the message. A message sent by *RPC* has a color of its sender object, while a message sent by *SEND* is given a newly created color. There are three cases in which an object receives a new message:

1. If the object is processing no messages, it immediately begins to process the new message.
2. If the color of the new message is different from that of the object, the message is put at the end of the message queue of the object.
3. If the color of the new message is the same as that of the object, the object must be waiting for a reply from a certain object. In this case, the current context of the object is saved and the object starts to process the new message, i.e., begins to execute its corresponding method. After the method completes its execution, the saved context is restored and the object restarts to wait for a reply.

Objects and messages with the same color can be regarded as a control thread or a process. The colored message model allows an object to recursively call itself, including the case of *super calls*.

As an example, let us give several implementations of a factorial function in G (see Figure 3.1). The first one (a) is of the purely recursive style. `%` denotes the received message, and `%[1]` denotes the first argument of the message (see Section 3.5). `@` denotes the object that is executing the method (*self* in Smalltalk-80). Strings without any prefix, such as `x` or `y`, are instance variables. In order to compute the factorial of, say 3, by this script, the following code suffices:

```
f = *fact(.create)
z = f(.compute, 3)
```

which will assign the factorial of 3 to the variable `z`. The next implementation (b) is that of creating a new object for each recursive call. The third one (c) is of the purely actor style, where the result of computation is explicitly sent by a message. Note that `&` denotes sending a message by *SEND*. The message `.compute` requires two arguments: the integer for which to compute the factorial and the receiver of the result.

```

script fact
  on .compute
    if %[1] == 0
      return 1
    end if
    y = @(.compute, %[1] - 1)
    return %[1] * y
  end on
end script

```

(a)

```

script fact
  on .compute
    x = %[1]
    if x == 0
      return 1
    end if
    f = *fact(.create)
    y = f(.compute, x - 1)
    return x * y
  end on
end script

```

(b)

```

script fact
  on .compute
    x = %[1]
    r = %[2]
    if x == 0
      & r(.result, 1)
    end if
    f = *fact(.create)
    & f(.compute, x - 1, @)
  end on

```

```

  on .result
    y = %[1]
    & r(.result, x * y)
  end on
end script

```

(c)

Figure 3.1. Three Implementations of Factorial

### 3.5. Expression and Statement

Expressions in G are not typed, and when they are evaluated, they yield a value of one word or values consisting of a sequence of words. Each word can be interpreted in various ways as was noted before.

The comma operator (,) concatenates the values of its operands. For example, the expression 1,2 yields two values 1 and 2. The comma operator can also be used in the left-hand side of an assignment. For example, x,y=y,x swaps the values of x and y.

The expression % denotes the message an object is processing. Messages are not vectors, but their components can be extracted by brackets. For example, %[0] denotes the message selector, and %[1] denotes the first argument of the message.

G has three kinds of variable: 1) *global*, 2) *instance* and 3) *temporary*. Their scope is 1) the entire program, 2) an object and 3) an invocation of a method, respectively. Since G does not have variable declarations, variables should be distinguished by their syntax, i.e., by their prefix: 1) \$, 2) none and 3) %.

The expression @ denotes the object in which it is evaluated. The expression @@ also denotes the object, but used for *super calls*.

The set of operators for building expressions is almost the same as that of the C language. The assignment operators and the conditional operator, however, are missing in G. Note that assignments are statements in G. The comma operator in G, explained at the beginning of this section, is not the operator for sequential evaluation, but the operator for concatenating



values.

Statements in G are constructed from primitive statements by combining them in the form of a compound statement. Primitive statements of G are: *expression*, *assignment*, *RPC*, *RPC-assignment*, *SEND*, *return* and *goto*. Note that the RPC construct can only appear as a statement, and not as a part of an expression. This is to simplify the implementation of the virtual machine. Compound statements of G are: *if*, *while*, *for* and *switch*. As is clear from the already given examples, statements in G are always separated by a newline.

### 3.6. Developing G code

The compiler of the G language resides in the virtual machine. This allows program-generated G code to be compiled instantly. G code in a file is loaded with the G-code loader called *gload*.

The methodology of debugging a program in a parallel object-oriented programming language is a subject that is still to be studied extensively in the future. However, for actually developing G programs, we had to prepare a debugging environment for G. Our debugger can stop a specified object and set a breakpoint on a specified method. When an object stops, the debugger can show the chain of RPC's ending in the current context of the object. It allows the programmer to evaluate expressions at an arbitrary context in the chain. It can keep more than one RPC chains, so the programmer can switch among the chains it keeps.

Source-level debugging is currently supported by connecting the debugger with the interface of GNU Emacs.

## 4. Experiences in Building Graphical User-interface

A number of experiences on parallel object-oriented programming were gained through the use of G. They include the problem of excessive parallelism, the importance of multiple-inheritance, etc.

### 4.1. Parallelism

Merging the two abstractions, object and process, reduces efforts of a programmer, because he or she can naturally consider each object as an active agent and no longer needs to explicitly create processes for gaining parallelism. In the conventional programming paradigm, in order to handle a menu, for example, a process should be explicitly created, which executes an input statement and waits for an event to pop up the menu. On the other hand, in the parallel object-oriented paradigm, nothing is required besides creating a menu. The object is already in the state of waiting for a message, i.e., an event, and an event automatically creates a control thread, when it reaches the object.

However, the purely parallel object-oriented programming paradigm has some defects because of its excessive parallelism. For example, when a group of objects send messages to one another by RPC, there always exists a danger of dead-lock. In such a case, the origin of an RPC chain should always be a unique object; this object should control all the objects in the group. When another object in the group wants to begin a job performed by some objects in the group, it should send a message to the controlling object by *SEND* to initiate the job. When an object outside the group wants to send a message to a group, it should

send a message to the controlling object. Programming styles as this one are not obvious for novice programmers, and they should be taught explicitly.

When a single job is performed by a group of objects, it is not always obvious how to initialize and terminate the objects in the group. Particularly, it is very hard to terminate the objects in a right order. If a right order is not followed, unexpected messages are sent to already killed objects.

In a purely parallel object-oriented model, all the information of an object is obtained by sending an appropriate message to the object. Even a type of an object is obtained by sending a message. However, when an object is busy in doing a job, or is blocked for a certain reason, no information can be obtained as to the object, because all the messages are queued and not processed.

In G, for avoiding such a situation, basic methods, which correspond to the methods of the *Object* class in Smalltalk-80, can be invoked without sending a message to an object. They can be invoked by sending a message whose selector is `.object_method` to the special object named `*interpreter`:

```
*interpreter(.object_method, object, message)
```

## 4.2. Multiple-inheritance

The facility of inheritance is one of the features that characterize object-oriented programming. While building graphical user-interface, we often find similar derivations in more than one chains of single inheritance. In such cases, multiple-inheritance allows one to describe the difference between a superclass and a subclass as a script (usually called a *MIXIN*), so that a single script suffices for those chains.

How to introduce inheritance into parallel object-oriented programming is one of the challenging issues in the study of object-oriented programming. In G, however, the colored message model allows an object to invoke its methods recursively, including the case of super calls.

## 5. Macro and Micro Stubs

The stub generator of GMW reads the stub description of an application, which corresponds to the so-called *application model* of a UIMS, and generates server stubs and client stubs with which a UNIX process behaves as a parallel object or a set of parallel objects. Note that stubs in G are completely compatible with the colored message model, so that processes connected through stubs can call each other recursively.

For each UNIX process with a macro server stub, an object called a *delegate object* is allocated in the virtual machine of GMW which represents the UNIX process in the machine. Messages to the delegate object are converted into packets and sent to the UNIX process, where the server stub decomposes packets and calls specified functions with appropriate arguments.

With micro server stubs, besides the delegate object representing the UNIX process, an object called a *micro delegate* is allocated for each object in the UNIX process. Messages to a micro delegate are also converted into packets and sent to the UNIX process, where specified operations are performed on the corresponding object in the UNIX process.

If client stubs are attached to a UNIX process, it can send messages to an arbitrary object in the virtual machine by calling appropriate functions in the client stubs.

### 5.1. Macro Stub

Currently, GMW supports a macro stub generator for C programs. A stub description is of the form:

```
A {
    F1(input types; output types)
    F2(input types; output types)
    ...
}
```

*A* is the name of the application. *F<sub>i</sub>*'s are the names of the methods served by the application. For each method *F<sub>i</sub>*, the application should prepare a function named *A.F<sub>i</sub>*. This function is called when a message whose selector is *.F<sub>i</sub>* is sent to the delegate object corresponding to the application. The input and output types specified for each method are used to determine the type of its corresponding function and to convert a packet into arguments for the function.

An actual application program consists of the functions served by the application and the main routine including the following code:

```
gmw_connect_UI(&argc, argv,
               "iconeditor_panel"l,
               "iconeditor/PANEL/ie_p.g"l, NULL);
gmw_process_message_loop();
```

The arguments to `gmw_connect_UI` are the command-line arguments including the name of the GMW server, the script name and file name of the user-interface part of the application and some additional data. `gmw_process_message_loop` begins a loop for reading messages and invoking functions, which corresponds to an event-loop in other window systems or toolkits. The loop calls the server stub code generated from the stub description and linked with the application program.

### 5.2. Micro Stub

Micro stubs are intended to allow an application process to have an indefinite number of application objects. In a circuit editor, for example, circuit elements are naturally implemented as objects in an application program written in C++ or Objective C. Such objects should have their own user-interface part in the UIMS. Micro stubs promote the separation of the user-interface part and the application body even at the level of application objects.

Currently, GMW supports a micro stub generator for C++ programs. A micro stub description is an extension of a macro stub description and is of the form:

```
A {
    F1(input types; output types)
    F2(input types; output types)
    ...
    subclass: C1(input types)
        M11(input types; output types)
        M12(input types; output types)
        ...
    subclass: C2(input types)
```

```

        M21(input types; output types)
        ...
    }

```

$C_j$ 's are class names and  $M_{jk}$ 's are methods for  $C_j$ . For each class  $C_j$ , the application should prepare a C++ class  $A\_C_j$ , which should be derived from the automatically created class  $A\_super$  and have the initialization method  $A\_C_j$  whose type is determined by the input types specified after  $C_j$ .  $A\_C_j$  should also have methods  $M_{jk}$  whose type is determined by the input and output types for  $M_{jk}$ .

When a message whose selector is  $.C_j\_create$  is sent to the delegate object, a C++ object of  $A\_C_j$  is created and at the same time, a micro delegate corresponding to the C++ object is created in the virtual machine. Sending a message whose selector is  $.M_{ij}$  to the micro delegate calls the method  $M_{ij}$  of  $A\_C_j$  on the C++ object.

Since C++ objects are executed by a single UNIX process, however, they cannot run in parallel.

## 6. Related Works

A comprehensive survey on UIMS is in [6], where historical research systems as well as commercially important systems are described in detail. Here, we only consider some of the widely used commercial systems and compare their approaches with that taken by GMW from the viewpoint of actually developing user-interface programs.

The naive idea of separating the application body and the user-interface part of an application is in Macintosh Toolbox [2] and also in OS/2 Presentation Manager [8], where the presentation component of user-interface is created not as programs but as resources and can be easily modified without affecting the application body. In those systems, it is firmly recommended that programs be coded in the event-driven style, although no support tools are prepared.

Prototyper [10] is one of the first layout editors available as a commercial product. Windows and dialogue boxes in Toolbox can be created without programming. It generates a template for an event-loop, so that the programmer should only fill in comments in the template.

X toolkits such as Motif [9] have several advantages over Toolbox or Presentation Manager: 1) Callback functions can be specified for each kind of event, so that the programmer does not have to explicitly code the event-loop. 2) Widgets comprise a class hierarchy and one can easily derive one's own widgets from already existing widgets. 3) Graphical constraints among widgets can be implemented by geometry managers. X toolkits also have resource managers, and in Motif in particular, User-interface Language (UIL) is supported for describing resources.

NeWS [11] adopts the virtual-machine-server architecture, and allows entire user-interface code to be written in PostScript and run inside the window server. The communication between application programs and user-interface programs is, however, somewhat ad hoc; ctops only allows C programs to call PostScript programs. This means that the overall control is in the application body, though actual programs may be written in the event-driven style.

NextStep [12] is the most challenging system at the time we are writing this paper. The layout editor called *Interface Builder* can handle and edit programmer-defined application

objects as well as system-supported tool objects. The link between the sender and the receiver of a message can be specified interactively using icons. For each application, Interface Builder generates a project file, a main file and class files in Objective C, a user-interface file, and a makefile. The class files are templates and should be edited by the programmer as in Prototyper. In NextStep, however, the idea of separating the user-interface part and the application body is vague, and an entire application is usually implemented as a single UNIX process with a single thread of control.

## 7. Conclusions

This paper described a UIMS that is based on a parallel object-oriented model and supports a stub generator that generates macro stubs for C programs and micro stubs for C++ programs. The parallel object-oriented programming language G has proved the importance of parallel object-oriented programming for building graphical user-interface. An application built on the UIMS is naturally split into server processes integrated through the user-interface part running inside the UIMS.

## Acknowledgements

We would like to thank all the members of GMW and related projects as well as all the people who patiently wrote application programs on GMW.

## References

- [1] Anderson, D. B.: Experience with Flamingo: A distributed, object-oriented user interface system, *Proceedings of ACM OOPSLA-86 Conference* (1986), pp.177-185.
- [2] Apple Computer, Inc.: *Inside Macintosh, Volume I*, Addison-Wesley Publishing Company, Inc., 1986.
- [3] Birrell, A. D. and Nelson, B. J.: Implementing remote procedure calls, *ACM Transactions on Computer Systems*, Vol.2, No.1 (1984), pp.39-59,
- [4] Goldberg, A. et al.: *Smalltalk-80, The Language and its Implementation*, Addison-Wesley Publishing Company, Inc., 1983.
- [5] Green, M.: Report on Dialog Specification Tools, *User Interface Management Systems*, Pfaff, G. E. ed., Springer-Verlag (1985), pp.9-20.
- [6] Hartson, H. R., Hix, D.: Human-computer interface development: Concepts and systems for its management, *ACM Computing Surveys*, Vol.21, No.1 (1989), pp.5-92.
- [7] KABA Software Research Group: Overview of GMW+Wnn System, *2nd IEEE Conference on Computer Workstations* (1988), pp.170-177.
- [8] Petzold, C.: *Programming the OS/2 Presentation Manager*, 1989.
- [9] Rockwell, P.: OSF/Motif: consistency and elegance for users and application designers, 1989.
- [10] SmetherBarnes: *Prototyper*, 1987.
- [11] Sun Microsystems: *NeWS Preliminary Technical Overview*, 1986.
- [12] Webster, B. F.: *The NeXT Book*, Addison-Wesley Publishing Company, Inc., 1989.
- [13] Yonezawa, A., Mario, T.: *Object-oriented Concurrent Programming*, The MIT Press, 1987.





**Masami Hagiya**  
*Kyoto Univ.*

Received B.S. degree and M.S. degree in information science from University of Tokyo in 1980 and 1982, respectively. Received Ph.D. degree in mathematical sciences from Kyoto University in 1988. Working for Research Institute for Mathematical Sciences of Kyoto University since 1982.



**Kouji Ohtani**  
*ASTEC Inc*

Received B.S. degree in electrical engineering from Kyoto University in 1983. Received M.S. degree in information science from Kyoto university in 1985. Working for ASTEC Inc. since 1985.

## MTX – A Shell that Permits Dynamic Rearrangement of Process Connections and Windows

Stephen A. Uhler

Computer Systems Research Division  
Bellcore  
445 South St.  
Morristown, NJ 07960  
sau@bellcore.com

### ABSTRACT

MTX is a UNIX<sup>™</sup> *shell*, implemented as a single user process, that has an expanded notion of pipes and I/O redirection. MTX supports creating and manipulating processes, connecting their inputs and outputs in an arbitrary directed graph. Each process can have as many windows or *virtual terminals* as required for user interaction. Through a control window, MTX can dynamically manipulate the connections among ongoing processes. In addition, MTX can rendezvous and connect to MTX servers on remote hosts, conveniently providing pipe connections across a network of computers. This paper describes the use and implementation of MTX on a network of UNIX workstations.

### Introduction

An important contribution of UNIX to computer science is the notion of a pipe [1]. A pipe enables the output of one process to be directly connected as the input of another process, as in `ls | wc`. The `ls` lists the files in the current directory; the `wc` counts them. Neither `ls` nor `wc` needs any prior knowledge of the other to work together. They only require a common format for data exchange: ASCII characters.

Although pipes are pervasive in UNIX, especially for text manipulation, they have several inherent limitations that, if mitigated, could make pipelines even more useful. The common UNIX shells, *sh* [2] *ksh* [3] and *csh* [4] support linear pipelines. Each process in a pipeline has exactly one input, and one output stream.

Although interprocess communication is limited to linear pipelines by the shell, UNIX supports a more general connection structure, that of a directed graph of processes and interconnections. Figure 1 shows an example of four processes connected in a nonlinear pipeline. There are at least two shells that remove the restriction of strict linearity of shell pipelines, *2dsh* [5] and *gsh* [6]. Although these two shells better exploit the UNIX interprocess communication primitives than do the common shells, because the command language used to specify the graphs of processes and interconnects is still linear, the command language is awkward. Setting up sophisticated graphs of processes is complex and cumbersome. To use the new connection facility effectively, processes must determine how many inputs and outputs they have, and what the semantics of the various streams are. Nonetheless, there are circumstances in which this type of extension to pipes is useful.

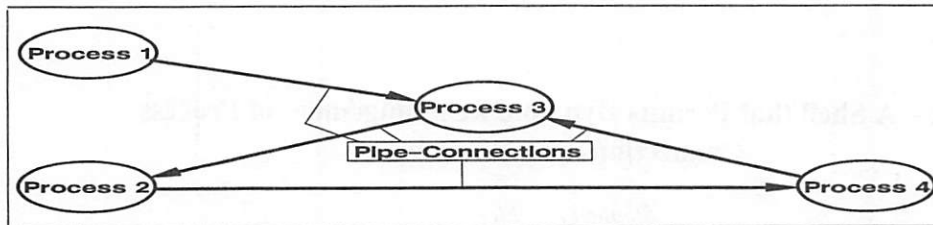


Figure 1. A directed graph of UNIX processes

UNIX also has the notion of a *terminal*, which is a channel for interaction with the user. Traditionally, a terminal is a physical device with a keyboard and a display. Normally there is only one terminal per user, so a pipeline of processes has at most one terminal. Because of this limitation, most processes that function effectively in a pipeline are data filters. They read their input, transform it based on some preset rules, then write their output. Once the process is started, no interaction with the user is allowed.

If each process in a pipeline could have its own interactive channel, or *virtual terminal*, it could prompt the user for information as it was running. Currently, a process must obtain its input from the command line. This would alleviate the burden of completely specifying the function of the command in advance. One could imagine a special version of *grep* that would permit the user to change the regular expression as *grep* was running. While not possible in the traditional UNIX timesharing environment, with a physical terminal per user, the emergence of window systems as the primary user interface to UNIX, makes it practical to provide as many *virtual terminals*, (windows with terminal semantics) to the user as needed. This presents the possibility of constructing a pipeline of interactive processes, as in Figure 2. By taking advantage of this capability, processes that participate in pipelines need not be restricted to passive data filters. They can be fully interactive and permit the user to change their operation dynamically without interfering with other interactive processes elsewhere in the pipeline.

By allowing processes to be connected in a directed graph, with a *virtual terminal* for each process, a much greater range of capabilities is available to the user. Along with this greater flexibility comes a greater complexity. Setting up the connections, processes, and windows properly can quickly get out of hand for all but the simplest of pipelines.

Fortunately, with the ability to use multiple windows, it is possible for the shell, which traditionally relinquishes control of the terminal as soon as the pipeline is started, to have its own window in which to maintain a dialogue with the user, even after starting a command. Through this shell window, connections among processes can be altered dynamically, once an extended pipeline has begun. New processes or connections can be added or removed from the pipeline as needed.

The advantages of dynamic reconfiguration of interactive processes has been demonstrated in the Synergy Dataflow System [7]. Interprocess communication in this system, unlike UNIX pipelines, was designed for programs written specifically to take advantage of the connection manipulation capabilities. It is implemented using new system calls as enhancements to the UNIX kernel, which are understood by the programs that use them. It does not address the ability to integrate existing programs into this environment, and consequently can not provide an environment compatible with existing application programs.

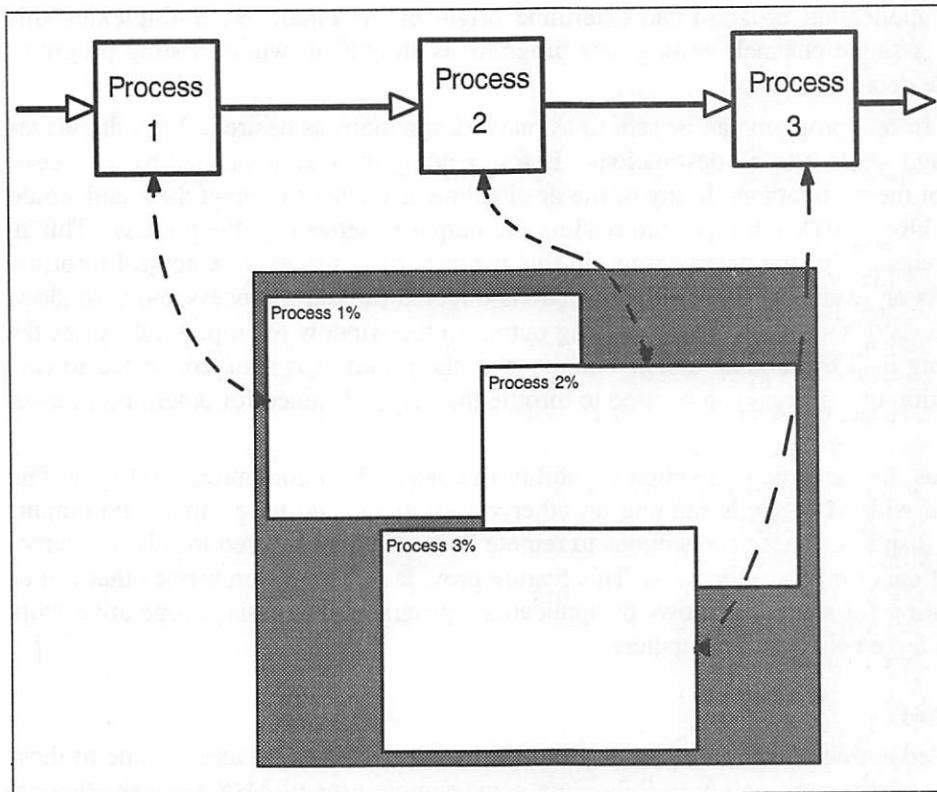


Figure 2. An example of an interactive pipeline

### What is MTX

MTX is a UNIX *shell* that has an expanded notion of pipes and I/O redirection. MTX supports: 1) the creation of processes whose inputs and outputs can be connected in an arbitrary directed graph, 2) the creation of as many *virtual terminals* as required by the processes, 3) the dynamic manipulation of the processes and connections after the processes have been started and, 4) the rendezvous and connection to MTX servers on remote hosts, which provides pipe connections across a network of computers.

Ordinarily, MTX gives a process two pairs input-output (I/O) channels. The first pair is for *stdin* and *stdout*, which are normally used by UNIX shells for pipe connections. The second pair of I/O channel is obtained by reading from */dev/tty*, and by writing to *stderr* or */dev/tty*. This is the interactive connection to the user. Many existing UNIX programs already follow this convention, and thus can be used unchanged with MTX. For non-interactive programs such as data filters, or for programs that use only one I/O channel, the two channels can be combined, with the process seeing a familiar terminal interface. The common command shells and editors are in this category. A separate window, belonging to MTX, allows the user to control and modify the state of the process and window connections, and to add or remove processes, windows or connections as needed.

When multiple input streams are directed to the same process, their inputs are merged onto the same stream. Processes do not have to know how many inputs they have, or which file descriptors they need to read. Consequently they need not deal with the complexities of retrieving input from multiple places. This avoids the problem of waiting for input on one input stream while data accumulates on another. Existing programs that deal only with one stream still function properly. An application that is designed to deal with multiple input streams, instructs MTX to put a marker in the data stream whenever input arrives from a different source. Upon reading

this marker, the application program can determine origin of the input. By multiplexing this information over a single channel, writing new programs is simplified, while existing programs exhibit reasonable default behavior.

The output from a program can be sent to as many destinations as desired. MTX duplicates the data stream and sends it to all destinations. Before reading the data generated by a process, MTX checks all of the destinations. If any of the destinations is unable to accept data, and would cause a *write* to block, MTX refrains from reading the output generated by the process. This in turn causes the generator of the data to stop. In this manner, MTX passes flow control information upstream. As an example, if the output to *cat* is directed to both a process and a window, and the user keys *CNTL-S* in the window, causing output to the window to stop, MTX causes the *cat* to stop writing data by sending a *stop* command to the *pseudo-terminal* connected to *cat*. Thus any destination of a process can be used to throttle the output destined for other processes or windows.

MTX extends the dynamic redirection capability to a network of computers. MTX can find and communicate with MTX shells running on other workstations, and have inputs and outputs routed to remote displays. These connections to remote hosts can then be used locally as sources or destinations of data for local processes. This feature provides a connection service that can be used as a foundation for shared windows or applications programs that assist cooperative work spread out over a large network of computers.

### Sample Uses for MTX

The expanded notion of I/O redirections provided by MTX has several uses. Some of these uses are obvious, others more obscure. Below are some sample uses of MTX for tasks that are difficult, if not impossible, to achieve with an ordinary UNIX shell.

As a simple example, MTX can be used to keep a transcript of a terminal session. Since MTX permits any stream to be duplicated, all input from the user can be saved in a file as it is sent to a program. Similarly, all output from the program to the user can be duplicated and saved as well, either in the same file or a different one. If desired, the program transcript can be filtered before it is saved in a file, perhaps to remove superfluous output, or to time stamp the data. Transcribing can be started or stopped at any time by adding or removing a connection to a running process.

MTX can be used to capture output from a command. It is common to issue a command, then to realize valuable output is scrolling off the display, such as listing a file with *cat* that was longer than you expected. The output would be nice to keep, if only to browse in a more leisurely manner. A typical solution is to terminate the command, and start it over, only with its output directed to a file or into a file browser. Unfortunately, sometimes it is expensive to restart a command, either because it will take a long time to reach the point of output generation, or because the process changes its environment as it runs, so recreating the initial state of the program is not feasible. By using MTX, the output of the program can be temporarily suspended, its output duplicated and sent either to a file or to another program. In an MTX environment, it is possible to have a browser available, so output from an arbitrary program can be connected to it.

A somewhat more obscure, but still useful application of the current implementation of MTX is that of a session multiplexor. Often it is only possible to have a single connection to another machine. This is the normal case with dial-up lines. Ordinarily a program such as *tip* [8] or *kermi* [9] would be used to set up the connection, but then only a single window on the local host can be used on the remote computer. However, if MTX is run on the remote computer after *tip* is used to start a connection, MTX can create multiple windows on the local host, with one or



more for each process running on the remote host, as shown in Figure 3. The data and commands for the processes are multiplexed over the same connection, thus providing a multi-window environment using a single byte stream connection. MTX derives its name from this capability, that of **M**ulti**T**iple**X**ing multiple process streams.

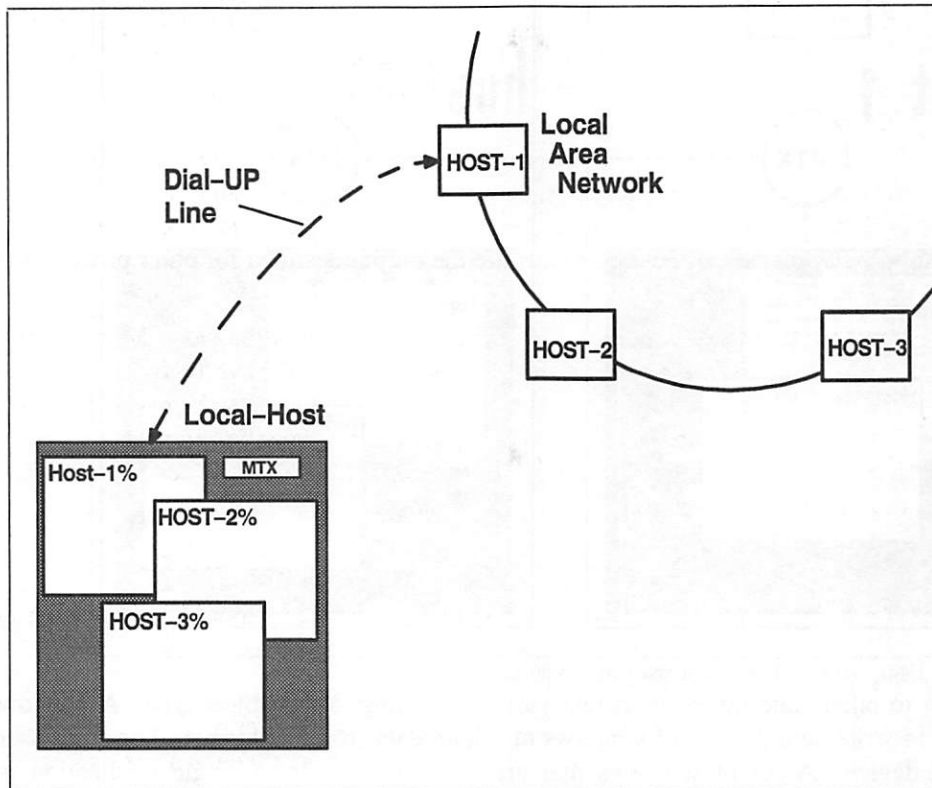


Figure 3. Using MTX to multiplex multiple processes over a single data channel

Perhaps the greatest potential use of MTX is shared windows. A user can connect to an instance of MTX running on different workstations, and send output to it, or receive input from it. An illustrative example is multiple users sharing an editor in different places. A user can start a single instance of the editor, with its output going to several windows, one on each computer. Similarly, input from one or all of the computers can be sent to the editor, permitting simultaneous editing of the same file by several users. This is shown in Figure 4, with the editor *vi* on *host1* being shared by two users, one on *host1*, the other on *host2*.

The above example demonstrates a simple use of sharing; there are no facilities to control or moderate the shared session. The shared application, in this case an existing editor, was not written to understand the notion of sharing. The MTX facilities allow for programs to interrogate and control their connections so new applications can be built that make intelligent use of this capability.

### User Interface Description

To understand the user interface, it is necessary to understand the terminology used by the current MTX implementation. MTX understands two basic entities, objects and connections. An object has two communicating ports, one for input (to the object) and the other for output (from the object). MTX understands three types of objects: 1) windows, 2) processes, and 3) ports to remote hosts (or just hosts). All objects have a name, either chosen by the user, or picked by

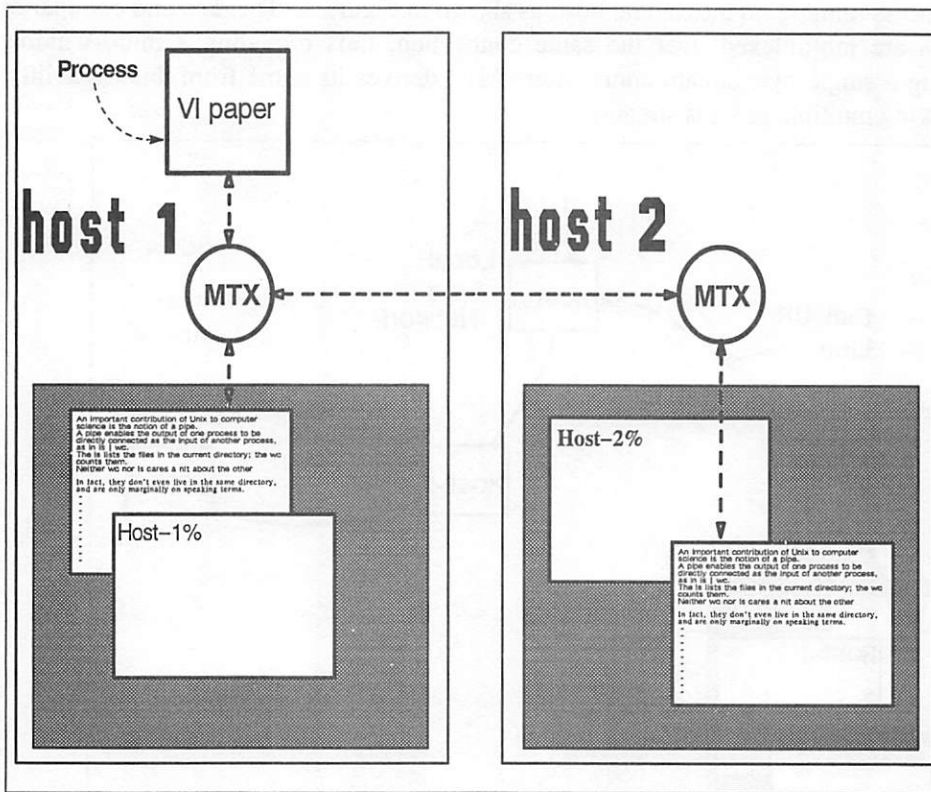


Figure 4. Using MTX to share a processes on two hosts

MTX, in addition to other state information that varies according to the object type. A window object is needed to write data to a set of windows or obtain data from the keyboard or other user controlled input device. A set of windows that are all controlled by a single application is accessed through a single window object. Similarly, a process object is the communication port to a process. Each process has one or two process objects, one for file descriptors 0 and 1, the other for file descriptor 2. The process sees a *virtual terminal* interface for each of its process objects. Finally, host objects are used for connecting to MTX servers on remote hosts. There are one or more host objects for each remote host, one object for each data stream. A host object is connected, via a stream socket, to a host object on the remote MTX.

Data flows between objects through a unidirectional data channel called a connection. Each object can have many connections; an object can even be connected to itself. Every connection must start at the output port of one object and terminate at the input port of another. The user interface to MTX consists of commands that manipulate the objects and their connections.

When MTX is started, it configures a control window on the display. This control window has an input line for typing commands to MTX, a message line for status information and messages, and a title line to identify MTX. Most of the user interaction with MTX is through a hierarchy of pop-up menus associated with the control window. The primary entries on the root menu are for creating windows and processes; displaying, adding, or deleting connections; and connecting to or displaying the names of remote hosts. The other entries exist to manipulate various debugging and configuration information, as well as exit or suspend MTX. Selecting an item on the root menu invokes the default action for the submenu anchored at that item. Figure 5 shows a sample MTX session taken from a portion of the window system display on the host *snoozy*. In this example two users, one on host *snoozy* and the other on host *zippy* are sharing a single instance of the editor *vi*. A command script created the *vi* process (shared-proc), a pair of

windows, a channel to a remote host (zippy), and all the connections. The smaller window (named *monitor*), displays the keystrokes typed by the user on *zippy*, whereas the *edit* window appears with identical contents on both hosts. The MTX connection window displays the connections among the objects.

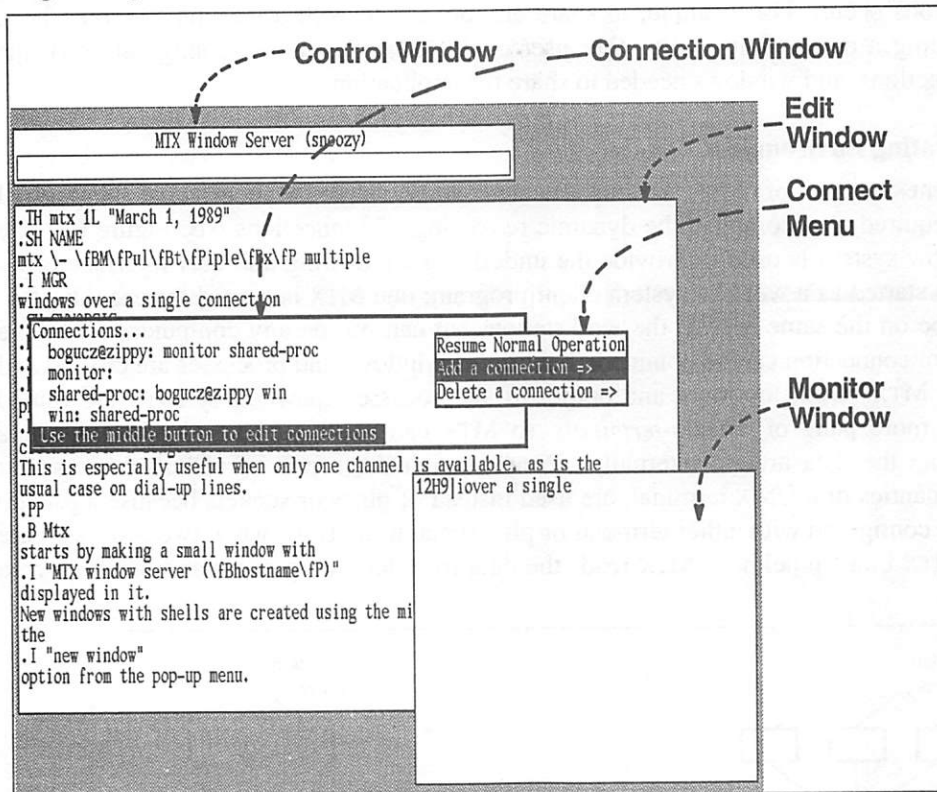


Figure 5. Sample display of MTX running

The default action for the root menu is to create a process (usually an ordinary shell), a window, and a pair of connections. This configuration behaves exactly like shell connected to a terminal. The options on the sub-menu include starting a process or window with no connections. The connections are added later, either through the connection modification options, or with a command script. Provisions are available for users to customize the submenus, adding their own entries.

The user can view and change the current connection information by selecting the *change connection* entry of the root menu. When selected, a configuration window pops-up. The name of each object is displayed along with the other objects connected to it. Pop-up menus associated with the configuration window can then be used to add or delete connections.

The *host* entry on the root menu has a submenu that lists the remote hosts on the network that are available for connections. If a remote host is selected, both the local and remote MTXs create an object for the other host. That *host* object will then appear in the connection window, and can be either the source or destination of a connection. If the remote MTX initiates the connection to the local MTX, a local host object is created. A message is displayed on the status line notifying the local user of the connection. When a remote MTX server is created or destroyed, a message is displayed, thus informing the user of which remote hosts are available for connections.

In addition to the menu driven command interface, MTX supports a primitive command interpreter that allows known configurations of windows, processes and their connections to be set up automatically. The functionality of the command language will be extended as the need arises. This will probably include mechanisms to invoke command scripts automatically when certain connections occur. For example, to share an application with a user on a remote host, simply establishing a connection to the other user could cause the proper configuration of the processes, connections, and windows needed to share the application.

### The MTX Operating Environment

The current version of MTX is implemented as a single user process. No kernel modifications required to accomplish the dynamic re-routing of connections (see Figure 6). The MGR[10] window system is used to provide the underlying windowing and user interface capabilities. MTX is started as a window system client program; one MTX is started for each display. MTX need not be on the same host as the workstation, but can run on any computer to which a serial byte stream connection can be obtained. Additional windows and processes are created and manipulated by MTX. Both the input and output to all processes spawned by MTX are routed through one or more pairs of *pseudo-terminals* so MTX can maintain control of all the data streams and route the data around internally. Pseudo-terminals, a pair of virtual devices that simulate the semantics of a UNIX terminal, are used instead of pipes or sockets because a *pseudo terminal* can be configured with either terminal or pipe semantics. Thus, when two processes are connected by MTX in a "pipeline", MTX reads the data from the first process and forwards it to the second one.

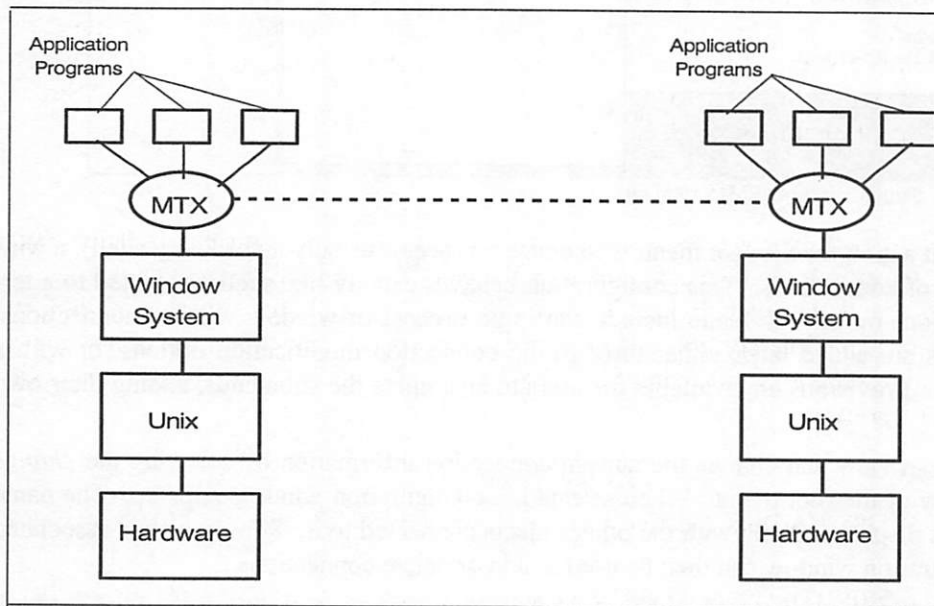


Figure 6. MTX relationship to other elements of the computing environment.

This extra step causes a decrease in the data throughput, compared to an ordinary pipe. Figure 7 shows an example of the performance decrease caused by running MTX. The tests are based on *catting /etc/termcap* to one or two windows simultaneously, and counting the number of characters per second written to a window (on a SUN Microsystems model 3/60).



MTX performance (characters/second)		
Test	MTX	No MTX
1 window	3700	4700
2 windows	1900	2400

MTX provides a terminal emulator in each window it creates for use by programs. Each window emulates the full window system protocol, so any window system application can run on top of MTX transparently (including MTX). Application programs need not be aware they are running on MTX.

In addition to implementing the window system protocol in each window, MTX provides extensions to the protocol as a means for programs to learn about their connection state. An application program can request to be informed by MTX when a connection to (or from) it is added or removed. MTX will provide the name of the object at the other end of the connection, as well as its type, and the type of its connection (input or output). By using this information, application programs can coordinate sharing among themselves, the users and other workstations. MTX also supports application queries for connection information. Applications that use these protocol extensions can still run directly with the window system, as the invalid protocol requests that would be handled directly by MTX are interpreted as unimplemented requests by MGR, and harmlessly ignored.

When MTX is started for a new display, it sends out a broadcast message to inform other MTX's running on remote hosts of its existence. Included in this message are the local user and hostname, as well as a port number to use to request a connection. When a user requests a connection to another host, a stream connection is made to the remote MTX, using the information contained in the broadcast message, and all data and control information are sent over this connection. By implementing MTX as a distributed service, the bottleneck that would result from a single MTX server on the entire network is relieved.

## Summary

MTX is a shell for Unix that permits the dynamic rerouting of pipe connections, the creation of multiple virtual terminals, and a more general mechanism for communication among processes than is typically found in the shell. In a network environment, MTX provides sharing of windows and processes among multiple computers and their displays. These capabilities permit the shell and other application programs to take advantage of the networking and windowing capabilities of modern UNIX systems.

## References

- [1] Ritchie, D. (1978), "A Retrospective" *Bell System Technical Journal*, 57 (6), pages 1947-1970.
- [2] Bourne S. (1978), "The UNIX Shell" *Bell System Technical Journal*, 56 (6), pages 1971-1990.
- [3] Korn, D. (1983), "KSH - a Shell Programming Language", *Proceeding of the Summer USENIX Conference*, pages 191-202.
- [4] Joy, W. (1980), "An Introduction to the C Shell", *UNIX User's Manual, Supplementary Documents*, Berkeley Ca.
- [5] Rochkind, M. (1980), "2dsh - An Experimental Shell for Connecting Processes With Multiple Data Streams", Unpublished Bell Laboratories technical memorandum.



- [6] McDonald, C. and Dix, T. (1988), "Support for Graphs of Processes in a Command Interpreter", *Software Practice and Experience* 18 (10) pages 1011-1016.
- [7] Haeberli, P. (1986), "A Data-flow Environment for Interactive Graphics", *Proceeding of the Summer USENIX Conference* pages 419-428.
- [8] Karels, M. and Leffler, S. (Ed.) (1984), *Unix User's Manual Reference Guide*, Berkeley, Ca.
- [9] Da Cruz, F. (1987), *Kermit, a File Transfer Protocol*, Bedford Mass: Digital Press
- [10] Uhler, S. A. (1987), "MGR - C Language Application Interface", Unpublished Bellcore technical memorandum.



Stephen Uhler  
Bellcore

Stephen Uhler joined Bell Communications Research at its inception in 1984, where he is a Member of the Technical Staff in the Computer Systems Research division. He has worked on computing environments and user interfaces for much of that time, and is the author of the MGR window system. Before joining Bellcore, Stephen was a Member of the Technical Staff at Bell Laboratories in Whippany N.J. where he worked on user interface management systems. Prior to Bell Laboratories, Stephen investigated engineering workstation computing environments at Exxon Research and Engineering Co. He received a M.S. degree in Chemical Engineering from Case Western Reserve University.

# USING UNIX AS ONE COMPONENT OF A LIGHTWEIGHT DISTRIBUTED KERNEL FOR MULTIPROCESSOR FILE SERVERS

*David Hitz, Guy Harris, James K. Lau, and Allan M. Schwartz*  
of  
*Auspex Systems*

## ABSTRACT

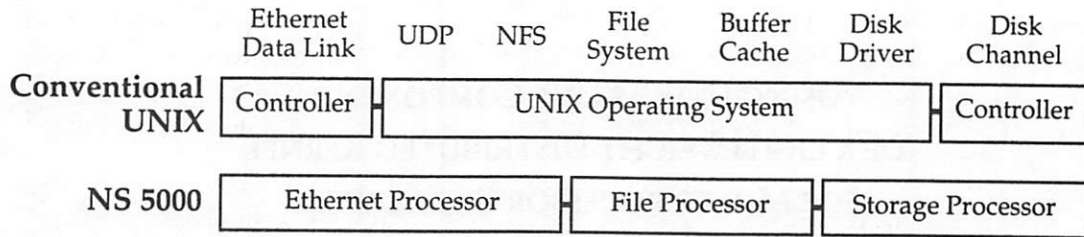
Auspex builds fast NFS file servers designed to satisfy the I/O demands of large networks and high-performance workstations. The architecture handles NFS operations quickly and efficiently by *completely eliminating* UNIX from the normal path of NFS service. We designed a message passing kernel that allows a slightly modified UNIX kernel to execute as a peer processor with Ethernet processors, filesystem processors, and disk storage processors. These non-UNIX processors respond efficiently to NFS requests and perform IP packet routing. A separate host processor running SunOS 4.0 provides full UNIX compatibility by servicing less time critical and less frequent requests such as Yellow Pages. Our message passing kernel is small (15 kbytes of object) and fast (10,000 messages per second into a 68020) and provides source code debugging for all processors.

## 1. INTRODUCTION

Today's NFS servers are hard-pressed to meet the demands of large networks populated with new high-performance client workstations. As client-server computing enters its midlife, client workstations are severely constrained by server I/O limitations. This I/O performance gap has developed because dramatic jumps in microprocessor performance have not been matched by similar boosts to server I/O channel performance. UNIX workstation vendors have traditionally designed servers by repackaging workstations in a rack, but adding larger disks, more network adaptors, or extra memory does not resolve basic architectural I/O constraints; neither does adding CPU MIPS.

At Auspex we have designed and built a high performance NFS file server called the NS 5000 with a Functional Multiprocessing (FMP) architecture that distributes NFS protocol processing across several highly intelligent processors [Auspex89]. In a nutshell, Auspex's Functional Multiprocessing architecture removes UNIX from the normal NFS processing path, using it instead to provide compatibility with standard UNIX services such as Yellow Pages and system administration. It optimizes a file server's most common actions—NFS operations—just as RISC processors optimize a CPU's most common instructions.

This diagram compares the distribution of NFS processing on the NS 5000 with that of a normal UNIX server.



In more detail, the processors perform the following functions:

- ◆ One UNIX host processor (HP) runs SunOS 4.0 to handle all non-NFS services. This provides complete compatibility with standard UNIX NFS servers but is not part of the normal NFS path.
- ◆ One or more Ethernet processors (EPs) receive and process Ethernet packets from the network. For NFS packets, IP, UDP, XDR, RPC and NFS layers are all handled locally, as is packet routing. EPs forward all unsupported protocols to UNIX.
- ◆ One or more file processors (FPs) manage local filesystems. FPs service file system requests from the UNIX host processor as well as those from the EPs.
- ◆ One or more storage processors (SPs) control SCSI disks and tapes. Each SP contains 10 parallel high-speed SCSI channels and provides driver-level services such as elevator sorting and retry.

The architecture is fast because the processors are designed specifically for NFS file service. It scales well because it supports multiple instances of each processor.

A key feature of the NS 5000 is a simple, high-performance message-passing kernel that provides a common development base for all non-UNIX processors. This Functional Multiprocessing Kernel (FMK) provides services such as lightweight process scheduling, message passing and memory allocation. Each processor executes an instance of FMK in its local memory and passes messages over the backplane. Some key features of FMK include:

- ◆ Very small size—less than 15 kilobytes of object code.
- ◆ Fast context switching—over 20 thousand / second on a 68020.
- ◆ Fast message-passing between processors—10 thousand / second into a 68020.
- ◆ Full support for *dbx* source code debugging.

Making the NS 5000 look like standard UNIX requires the host processor to integrate completely with the other processors. With FMK this is possible because the host processor looks just like another FMK peer processor in the system. The host processor provides UNIX compatibility services; other processors provide NFS services.

The remainder of this paper explores the design and implementation of FMK, the work required to integrate FMK and UNIX, and a more detailed description of how FMK and UNIX interact in the NS 5000 architecture.

## 2. MODEL FOR A FUNCTIONAL MULTIPROCESSING KERNEL

Most of the code running on the NS 5000 comes from UNIX. The TCP/IP protocol stack and Fast File System come from BSD 4.3, and we license NFS from Sun. All of this code requires UNIX kernel support, but running UNIX on all of our processors—even a stripped down UNIX—would be cumbersome and ill-suited to our needs. We wanted the smallest, fastest kernel that would satisfy our needs: a reduced primitive set kernel, if you will.

FMK is the result. FMK is a kernel for operating system development, not application development, and as such it provides a minimal set of fundamental services including lightweight process scheduling, message passing, memory allocation, simple timer services, and interrupt handling. It does not support memory management, process destruction or a robust user interface. The original specification provided just 16 primitives, and although it has now grown to thirty-something, we still consider it lean.

### 2.1. Overview of Model

FMK supports lightweight processes that communicate via synchronous (*i.e.* blocking) message passing primitives. Each process is identified by an FMK process-id or PID and consists of little more than a stack, a thread of control, a scheduling priority, and a queue of messages that have been sent to it. These processes are similar to, though simpler than, UNIX processes running in kernel mode. As with UNIX kernel processes, no preemption is allowed. A process continues to run until it explicitly gives up the CPU. Nothing in FMK precludes preemption; however, since most code running under FMK comes from the UNIX kernel we would have had to modify that code or disable preemption anyway. It seemed more sensible to omit it.

Message passing works identically whether the destination process is on the same processor or a different one. This makes it easy to tune the system by moving a particular process from one processor to another. As an example of this flexibility, we developed the file processor code under FMK on UNIX before moving it to its own processor.

Processes on the same processor share the same address space, while those on different processors do not. When a group of processes share data structures in their common address space, as opposed to communicating exclusively through messages, they may be moved to another processor only if they are kept together. Of course, processes that manage hardware such as Ethernet chips or SCSI channels must run on the processor with that hardware.

An FMK message consists of 128 bytes of data, the last few bytes of which are reserved for the kernel. The data always starts with a message type that specifies the operation requested of the receiving process. All messages have an associated C structure that defines their format. The FMK part of the message includes a pointer for queuing the message into linked lists, source and destination PIDs, and the like.

### 2.2. Fundamental Primitives

The following list summarizes the most frequently used FMK primitives. All primitives begin with “*k\_*” to reduce naming conflicts. The “*k*” stands for kernel.

- ◆ *k\_register*(name)      Assign the specified name to the current process.
- ◆ *k\_resolve*(name)      Return the PID of the process with the specified name.

- ♦ *k\_alloc\_msg()* Return a newly allocated message.
- ♦ *k\_free\_msg(msg)* Free the specified message.
- ♦ *k\_send(msg, pid)* Send message to the specified process. Block till it returns.
- ♦ *k\_receive()* Receive message sent to this process. Block if none ready.
- ♦ *k\_reply(msg)* Return message to process that originally called *k\_send()*.

The code below shows how these primitives can be used. The *get\_time()* function sends a message to a TIMED process that handles simple timer services. The *timed()* function implements the TIMED process. Note that *timed()* is intended to run in UNIX user space and uses standard UNIX system calls to get and set the time.

```

struct time      /* Ask the TIMED what time it is. */
get_time()
{
    struct time    time;
    struct time_msg *msg;
    K_PID          pid;

    msg = k_alloc_msg();
    msg->type = GET_TIME;

    pid = k_resolve("TIMED"); /* Find "timed" process. */
    msg = k_send(msg, pid); /* Send message, await reply. */

    time = msg->time;
    k_free_msg( msg );

    return time;
}

timed()          /* Implementation of TIMED process. */
{
    struct time_msg *msg;

    k_register("TIMED");
    while (1) {
        msg = k_receive();

        switch (msg->type) {
            case GET_TIME: gettimeofday(&msg->time, 0); break;
            case SET_TIME: settimeofday(&msg->time, 0); break;
        }

        k_reply(msg);
    }
}

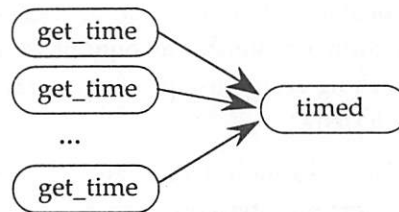
```

We can represent this message passing relationship in a graph like this:



Boxes represent processes, and the arrow represents a message being sent. Since TIMED could service multiple clients, the picture really looks like this:





There is just one TIMED process so only one GET\_TIME or SET\_TIME message can be handled at a time. For many services, this is desirable. For instance, requests to a DMA service process *should* be queued for sequential processing because a DMA channel can handle only one request at a time. On the other hand, messages to a filesystem service process must not back up just because the current message is waiting for disk I/O.

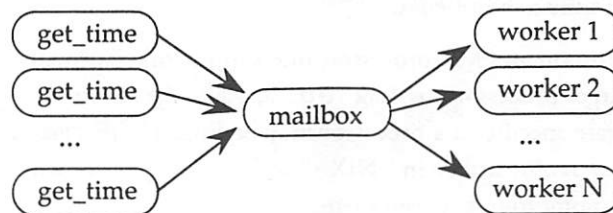
FMK provides mailboxes to allow more than one process to service messages sent to a particular PID. Sending a message to a mailbox is identical to sending a message to a process. Typically the creator of a mailbox also creates several worker processes to receive messages from it using

- ♦ `k_create()` Create a new process.

The arguments to `k_create()` include the address of a function to run in the new context, the size of the stack required for that function, a scheduling priority, and one argument for the new process. These processes receive messages from the mailbox using

- ♦ `k_receive_mbox(id)` Receive a message from a mailbox. Block if none ready.

If `timed` were reimplemented using mailboxes, the graph would become



Most services implemented in the NS 5000 use this model.

The examples above give a flavor of development with FMK. FMK also includes primitives to check whether messages are available without blocking, primitives to wait for interrupts and to signal their occurrence, and primitives to adjust the scheduling behavior of processes. These are less frequently used, but are required to provide particular services or to improve performance in special cases. FMK has also incorporated the UNIX kernel memory allocation functions from the BSD 4.3 Tahoe release with some modifications for supporting multiple memory types [McKusick88].

### 2.3. Omissions from FMK

Having reviewed what FMK offers, it makes sense to examine what it does not. We believe that FMK is more notable for features it omits than for those it includes. In particular, there is no mechanism for killing an FMK process; once a process is created, it lives forever. Processes cannot send messages asynchronously; they must wait for the reply. Processes can wait for only one event at a time; there is nothing like UNIX's `select(2)`. Finally, FMK has no memory management.

Each of these features would require extra processing in the low-level code that handles process scheduling and message delivery. FMK is fast in large part because of these simplifying assumptions.

In UNIX, leaving out these features would be painful. Processes are expensive and context switching is slow. Every process is expected to perform a reasonable amount of work. On the other hand, when processes are cheap and context switching rapid, groups of processes can perform functions that would be handled by a single process in UNIX.

These restrictions are not an exercise in masochism. Code based on a group of processes built from simple primitives can be simpler and more provably correct than code based on fewer processes with more complex primitives.

These principles are by no means new to FMK. Most of the fundamental ideas were originally developed in Thoth, from which the V kernel and Port are also derived [Cheriton79, Cheriton84]. Unlike FMK, which has no memory management, these kernels all support memory management models that allow processes to have private address spaces. They also allow process destruction. FMK has made some subtle but important changes in the relationship between processes and messages that have eased its integration with UNIX, but perhaps more importantly, FMK has carried the idea of a very small and very simple set of primitives to an almost obsessive extreme.

### 3. THE FMK PROGRAMMING ENVIRONMENT

We believe that time spent building a high quality development environment and good debugging tools is more than repaid in reduced development and debugging time and improved product quality. Our work focussed on making important UNIX tools like *dbx*(1) and *prof*(1) work with code running under FMK on the non-UNIX processors and on developing new tools where useful. For FMK in UNIX user processes, standard development tools work fine. For UNIX kernel debugging *kadb*(8) is archaic, but we haven't had time to write *kdbx*.

To compile FMK code to run on an FMK processor, one simply links normally compiled code with the *fmk.o* file for the target processor (*i.e.* *fmk\_ep.o*, *fmk\_fp.o*, or *fmk\_sp.o*). Code that does not depend on hardware specific to a processor may be linked with *fmk\_unix.o* and tested under UNIX. A new *ax\_startup*(8) utility in UNIX ("*ax*" for Auspex) downloads images to each processor and initiates communication between them.

#### 3.1. Development Tools

Running *dbx* on a peer processor from UNIX is also simple. The image is downloaded using *ax\_dbx* instead of *ax\_startup*. As with ordinary *dbx*, one starts the program by typing *run* and stops it with control-C. Single stepping, breakpoints, and tracing all work as expected. The only noticeable difference is that printouts from the processor come out on its console, not from *ax\_dbx*.

Another program named *ax\_util* provides a collection of services related to the various processors. It can collect core files (interpretable by *ax\_dbx*) or profiling data (interpretable by *ax\_prof*), and can also read and write a processor's local memory. The NS 5000 is designed to collect a core file from each processor in the case of a panic, although we don't expect this ever to happen except in carefully controlled test situations.

An interactive shell process runs under FMK. It supports several commands such as one that shows all FMK processes and their states and another that enables verbose tracing from FMK. Custom commands can be added for individual processors. The Ethernet processor, for instance, supports a command that shows network Mbuf statistics for each interface.

Finally, *ax\_util* can provide a virtual console connection from UNIX to any other processor. As an example of the power of this feature, one can *telnet* to an NS 5000, establish a virtual connection to any processor, and issue any command its shell supports. Of course, one must be super-user to do this, and some commands may be dangerous to the health of the system, but for debugging this feature is wonderful.

### 3.2. A Library of Standard Services

Although FMK itself is very lean, we have developed a library of standard functions and processes that provide useful services.

An FMK process that registers the name AX\_ERRD provides a link to the UNIX *syslogd*(8) message logging service. From any processor, one can send a message to AX\_ERRD to have a message printed on the UNIX console or appended to an administrative log file.

An AX\_TIMED process running under UNIX provides real time clock services to all processors, and can also be instructed to notify a processor when UNIX's time gets changed.

The library also includes the simple shell, some standard functions from libc such as *bcopy*(3) and *printf*(3), and FMK implementations of UNIX kernel functions such as *sleep*() and *wakeup*().

## 4. EXTENDING FMK TO COEXIST WITH UNIX

As an essential step in making our Functional Multiprocessing architecture look like standard UNIX, FMK must connect UNIX with the other processors. UNIX must integrate with FMK as a compatible peer processor.

Minimizing modifications to SunOS is also important because UNIX implementations are a moving target and we expect Sun and others to continue providing new and improved versions of UNIX in the foreseeable future. We want to smoothly track these new releases.

To implement services like a device driver interface to the storage processor, FMK primitives are provided within the UNIX kernel. On the other hand, we also wanted to support FMK primitives in UNIX user level processes because it is so much faster and easier to debug code there than in the kernel.

Two differences between UNIX and FMK make this integration difficult. First, UNIX processes can die whereas FMK processes must live forever. Second, FMK provides quick context switching for its processes, so interrupts typically wake up a service process. UNIX context switching is slower, so many device driver services are performed in the bottom half of the kernel at interrupt level where there is no process context and blocking is forbidden. It is difficult to provide FMK services in the bottom half because FMK requires processes to block in order to send and receive messages.

The general solution to these problems is to observe that the procedural interface to FMK under UNIX need not look exactly like that on the other processors as long as messages to and from UNIX look *just as if* they had been sent to standard FMK lightweight processes. On the other hand, we tried hard to minimize the differences between FMK on UNIX and FMK on other processors to make it easier to move code back and forth between the two and to reduce general confusion.

#### 4.1. Handling UNIX Process Destruction

We solved the process destruction problem by waving our hands. We observed that while UNIX process-ids change often, there are only a small fixed number of `user` and `proc` structures in the kernel. We reasoned that processes aren't really destroyed in UNIX, their contents and process-ids just change.

While this observation may sound fatuous, in practice it allowed the system to work. When UNIX boots, each UNIX process slot gets a unique FMK PID, and it retains that FMK PID even as the UNIX PID for the process slot changes.

This approach introduces two possibilities for error:

- ◆ If a UNIX process is killed during `k_send()`, the reply message returns to an empty process slot, or worse yet to a new process running in the same slot.
- ◆ If a UNIX process that has `k_register()`d a named service dies, the new process occupying that slot will almost certainly not know how to handle arriving messages.

We solved the first problem by making `k_send()` uninterruptible. A process in `k_send()` cannot be killed until after it gets its reply. Most UNIX processes that use FMK don't provide named services, and for them this solution is sufficient.

For processes providing named services, we made a simple rule: Don't die. To insure that such processes aren't killed we added a system call to UNIX to declare a process immortal—like `init(8)`. If that process dies, UNIX dies. There are three such service processes in the NS 5000.

#### 4.2. FMK in the Bottom Half of the UNIX Kernel

Providing FMK service in the bottom half of the kernel was harder. At this level UNIX provides no process context and blocking is not allowed. Yet the FMK message passing model requires a process context because it expects message senders and receivers to block.

One solution would have been to add lightweight processes that could run in the bottom half of the UNIX kernel. This didn't fit with our goal of keeping UNIX modifications simple and easy to port to new versions.

Instead we provided additional primitives to FMK that allow bottom half code to send messages without blocking and to receive messages at interrupt time. Although we dislike adding primitives to FMK, the requirements of interfacing with UNIX overrode that concern in this case.

### 5. FMK IMPLEMENTATION

For the non-UNIX processors, the implementation consists of two separate layers: a processor-independent FMK layer, and a processor-specific layer that handles hardware initialization and message passing between processors.

Although we have currently used the FMK layer only on 68K-based processors, it was written to be portable to other processors with the same byte ordering. The use of C structures to define message formats precludes processors with different byte orders. The cost of XDR-like format conversion for messages within our system would have been unacceptable.



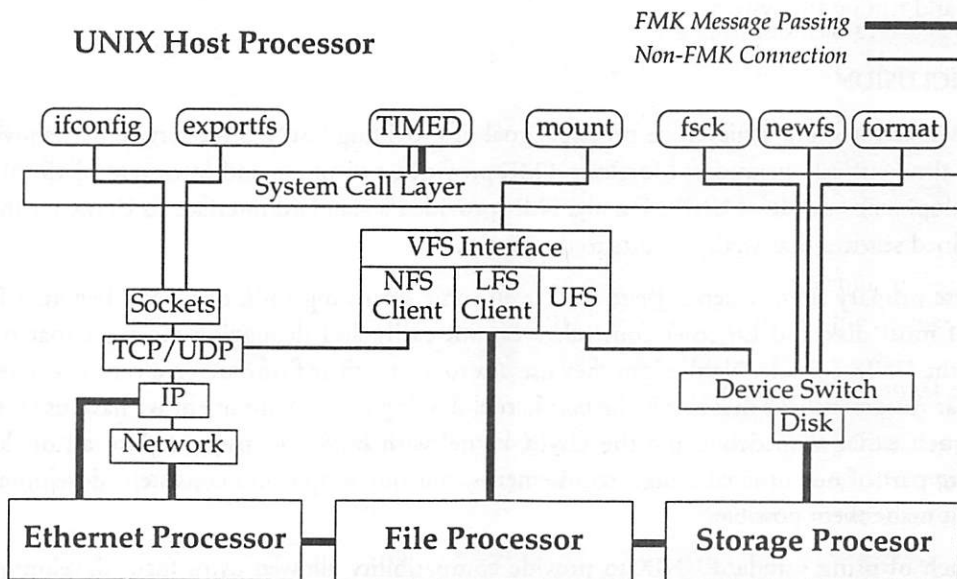
The hardware layer consists of additional code unique to the various processors. It supports not only FMK, but also features like DMA channels and RS232 console ports. The entire FMK kernel, including both FMK and hardware layers, compiles to between 12 and 14 kilobytes of object code for all non-UNIX processors.

Although the FMK primitives in UNIX are almost identical to those in other processors, the implementation is separate. Under UNIX FMK is implemented as a collection of functions that can be used anywhere in the top half of the kernel. We also added a device driver that provides access to the functions in user space. FMK user processes link with a special Auspex library to access these functions. This code should be easily portable to other BSD-derived UNIX implementations.

Since the design goal of the NS 5000 is to service many NFS clients, the ultimate performance measure for FMK is its ability to efficiently support large volume message traffic among peer processors. Between processes on a single processor, FMK can support up to 15 thousand k\_send/k\_reply pairs per second, and over the backplane FMK can deliver up to 10 thousand messages per second to one processor. The context switch time is under 50 micro-seconds. All of these number are for 20 MHz 68020 processors. Because the architecture was designed to remove UNIX from the standard data path, we did not optimize FMK so carefully for the host processor.

## 6. FMK IN THE NS 5000

Our goal in developing FMK was not to introduce yet another operating system into the world, but to build a tool for use in our file server. The diagram below shows how FMK connects the various components of the NS 5000. The very top of the diagram represents user space with the UNIX kernel in the middle and the three special purpose processors below. Additional processors can be added, but they have been omitted for simplicity.



In keeping with our strategy of modifying UNIX as little as possible, we put FMK interfaces at standard cleavage points: UNIX speaks to the storage processor through a device driver, to the file processor through a new VFS file system type, and to the Ethernet processor through the standard network "ip" driver.



The interface to the storage processor—implemented as a simple UNIX device driver—is the cleanest. The device driver for `/dev/ad*` (Auspex disk) simply converts incoming requests into FMK messages to the SP using the FMK primitives provided in the kernel. Since disk interactions are always initiated by top half UNIX routines, the extended FMK primitives for the bottom half of the kernel are never needed. This driver allows utilities like *fsck*(8), *newfs*(8), and *format*(8) which read disks directly to work unmodified.

The interface to the file processor is more complicated. To give UNIX access to filesystems that are mounted on a file processor, we created an entirely new filesystem type using Sun's VFS (Virtual File System) architecture [Kleiman86]. We call this new file system LFS, for Local File System [Schwartz89]. For each request through the VFS interface, it generates one or more messages to the file processor. LFS looks much like NFS, but with modifications to make operation over a local bus more efficient, and with additions to support mounting, exporting and quota control.

The interface to the Ethernet processor is the most complex. Unlike the SP or FP, the EPs must communicate with the bottom half of UNIX when packets using non-EP supported protocols arrive. An EP must also send messages to UNIX when IP addresses not in its route cache are encountered and to resolve keys for secure RPC. We made other modifications where UNIX was unable to handle the eight separate Ethernets.

Finally, there are many UNIX user level programs that use FMK to communicate with the other processors. The FMK service programs that provide timer and error logging services have already been mentioned. During development we also used FMK programs to query or control the Auspex processors. For instance, one program queries FMK to gather statistics such as the number of messages sent and the load average of that processor's CPU. These tools are not part of the standard UNIX interface because they refer to features not available in standard UNIX, but they are useful in developing and tuning the system.

## 7. CONCLUSION

Auspex's FMP architecture achieved its principal goal of providing fast NFS file service by removing UNIX from time-critical operations. Moreover, FMK provided a common and consistent platform for system development outside of UNIX. Finally, FMK provided a standard interface to UNIX for those nonstreamlined services that UNIX exports to the clients.

Beyond these primary achievements, there have been some surprising additional FMK benefits. The designers of most disk and Ethernet controllers cannot easily add debugging messages that print directly on the UNIX console. Neither can they use *dbx* to debug their firmware on a running system. Most of us at Auspex believe that this is the best kernel development environment we have used. It is certainly much easier than debugging the UNIX kernel with *kadb*, for instance. Not all of these features were part of our original design requirements, but our simple and consistent development environment made them possible.

Our approach of using standard UNIX to provide compatibility allowed us to focus development effort on the I/O subsystem, where the greatest improvements could be made. Our use of FMK and its environment allowed us to develop this code quickly and efficiently.

## 8. REFERENCES

- [Auspex89] Auspex Systems Inc.  
*An Overview of Functional Multiprocessing for Network Servers.*  
Technical Report 1, Auspex Systems Inc., October 1989.
- [Cheriton79] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager.  
Thoth, a portable real-time operating system.  
*Communications of the ACM* 22(2):105-115, February, 1979.
- [Cheriton84] David R. Cheriton.  
The V Kernel: A software base for distributed systems.  
*IEEE Software* 1(2):19-42, April, 1984.
- [Kleiman86] S.R. Kleiman.  
Vnodes: An Architecture for Multiple File System Types in Sun Unix.  
*Proceedings of the Summer 1986 USENIX Conference*, Atlanta, Georgia.
- [McKusick88] Marshall Kirk McKusick and Michael J. Karels  
Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel.  
*Proceedings of the Summer 1988 USENIX Conference*, San Francisco, CA.
- [Row84] John Row and David Daugherty.  
Operating System Extensions Link Disparate Systems.  
*Computer Design*, July 1984.
- [Row85] John Row.  
Lan Software Links Diverse Machines, OSs.  
*Mini Micro Systems*, September 1985.
- [Sandberg86] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon.  
*Design and Implementation of the Sun Network File System.*  
Technical Report, Sun Microsystems, January 1986.
- [Schwartz89] Allan M. Schwartz, David Hitz, and William M. Pitts.  
LFS—A Local File System for Multiprocessor NFS Network Servers.  
*Proceedings of the Sun User Group*, Anaheim, CA, 6–8 December 1989.  
Also Technical Report 4, Auspex Systems Inc., December 1989.

The following are trademarks of their respective corporations: Ethernet, Functional Multiprocessing, Functional Multiprocessor, FMP, NFS, NHFSStones, NS5000, SunOS, UNIX, VME



**Dave Hitz**  
*Auspex*

Dave Hitz is a member of Auspex's software engineering staff and has worked on Auspex's messaging kernel and filesystem software. Before Auspex Hitz worked at MIPS where -- among other things -- he ported the BSD 4.2 filesystem to System V.3 and optimized the underlying buffer cache. In previous jobs he slaughtered cattle and typed names onto Blue Shield Insurance cards. After dropping out of high school, he attended George Washington University, Swarthmore College, Deep Springs College, and finally Princeton University where he received his BSE in computer science in 1986.



**James K. Lau**  
*Auspex*

James Lau is manager of Auspex's networking group and has worked on Auspex's messaging kernel, network software and hardware. Prior to Auspex, he worked at Bridge Communications for over five years where he worked on TCP/IP, XNS protocols on variety of products. Mr. Lau received his BA degree in Applied Mathematics and Computer Science from U.C. Berkeley in 1981 and MS degree in Computer Engineering from Stanford in 1982.



**Guy Harris**  
*Auspex*

Guy Harris is a member of Auspex's network services groups and has worked on Auspex's NFS software, messaging kernel, and UNIX modifications. Prior to Auspex Guy worked at Sun where he memorized POSIX, X-OPEN, and SVID. At other jobs Guy memorized the version 6 and 7 kernels, and release III and V UNIX kernels, as well as all BSD releases. Guy received his S.B. in physics from MIT in 1975.



**Allan M. Schwartz**  
*Auspex*

Allan Schwartz is a member of Auspex's software engineering staff in the operating systems group. Prior to Auspex, Schwartz worked at Bridge Communications where he implemented various pieces of the Communications Servers and Network Control Servers. He fondly remembers hacking the Unix V6 kernel, a few years before. He attended Purdue University, receiving his BS in Computer Science in 1979 and Stanford University from 1979-1982. Schwartz is a member of the ACM.

# A Highly-Parallelized Mach-based Vnode Filesystem \*

Alan Langerman  
alan@encore.com

Joseph Boykin  
boykin@encore.com

Susan LoVerso  
sue@encore.com

Shashi Mangalat  
shashi@encore.com

Mach Operating System Project  
Encore Computer Corporation  
257 Cedar Hill Street  
Marlborough, MA 01752-3004

## Abstract

This paper describes the parallelization of a vnode-based filesystem layered on the Mach operating system. Encore parallelized portions of the Mach operating system's 4.3BSD compatibility code to increase performance. Subsequently, Carnegie-Mellon University (CMU) released a version of Mach with substantial modifications to the filesystem to provide NFS functionality. Using the framework left in place from our original parallelization effort, we migrated to the new vnode filesystem code by modifying our original locking policies to accommodate the new filesystem organization. Based upon performance analysis of our original version, the opportunity was also taken to revise the old locking policies to provide substantially increased parallelism. Details of the new locking policies and performance analyses are provided.

## 1 Introduction

The Mach operating system fully supports existing 4.3BSD programs by including large portions of the original 4.3BSD operating system code inside the Mach kernel [5, 8]. While the Mach portion of the kernel was designed for parallel operation on tightly-coupled shared-memory multiprocessors, the 4.3BSD code was originally designed for uniprocessors. Thus, Mach, as distributed from Carnegie-Mellon University (CMU), is forced to execute all the 4.3BSD compatibility code, as well as all interrupt handling, on a single processor, known as the *master* processor.

Encore is interested in Mach because of its multiprocessor support. In particular, Encore is developing a DARPA-sponsored 1,000 MIPS multiprocessor that will use Mach. As part of that

---

\*This research was supported in part by the Defense Advanced Research Projects Agency (DoD) through ARPA Order No. 5875, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-86-C-0158.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Multimax and UMAX4.3 are trademarks of Encore Computer Corporation. Unix is a registered trademark of AT&T Bell Laboratories.

NFS is a trademark of Sun Microsystems, Inc.



contract, Encore ported Mach to the Multimax, a symmetric shared memory multiprocessor using the National Semiconductor 32000 family of processors.

Experience with Mach on the Multimax showed that Mach delivered poor performance by comparison with Encore's commercial operating systems. Performance analysis showed that the majority of programs executed used the 4.3BSD compatibility code and therefore were restricted to run only on the master processor. As described elsewhere, we proceeded to parallelize portions of the 4.3BSD compatibility code, with emphasis on the filesystem and network[2]. Our approach of incremental parallelism was somewhat different than that of other scientists in the field[1, 7, 4, 3]. The resulting operating system delivered performance comparable to that of Encore's commercial Unix operating systems while providing a basis for future parallelization.

Encore obtained a pre-release copy of Mach version 2.5 that included a vnode-based filesystem, derived from Sun's *NFS* Release 3.2. To remain current with CMU we were confronted with the challenge of applying our inode-based filesystem parallelization changes to a vnode-based filesystem. The differences in functionality, data structures and code organization between the two filesystems are significant. Fortunately, these changes are bounded by the high-level system call interface, which remains largely unchanged, and the buffer-cache and low-level I/O routines, which also remain largely unchanged. Thus, we were able to retain our buffer-cache locking scheme, based on individual buffer and buffer cache hash chain locks, as well the low-level disk handling strategy.

The new vnode layer required us to design a new locking strategy. This new design allowed us the opportunity to incorporate our experience and performance analysis results from our original effort. As an example, the original implementation locked an inode from the time it was found in the inode cache until it was subsequently released. While this scheme had the advantage of reducing code modifications, it had the disadvantage of reducing parallelism; only one thread at a time could read entries in a directory, stat the contents of the inode, or read data from a file. The new locking strategy considerably enhanced parallelism for all vnode-based operations, with a small additional cost in terms of code modification.

The new filesystem showed improvements in overall parallelism as well as substantial improvements for parallel operations on the same directory or file. In particular, we observed performance gains for utilities that read the same directories and files. We also observed a considerable decrease in the time to translate pathnames that used common directories. We expect that the additional parallelism will have its greatest impact on file servers and databases that frequently reference the same files and directories.

We discuss the rationale behind the new locking strategy in Section 2. Locking strategies and problems are given in Sections 3, 4 and 5. Areas that have not been changed from our original effort are detailed in Section 6. Performance analysis is described in Section 7.

The current Encore Mach release, with parallelized 4.3BSD compatibility code and an inode-based filesystem, is known as Mach/0.5. Encore's edition of Mach version 2.5, with a vnode-based filesystem is currently in alpha-test and is known as Mach/0.6.

## 2 Background

As already stated, our effort to parallelize a vnode-based file system was based upon our previous experience parallelizing the 4.3BSD inode-based file system. The same locking strategy used in our previous work could have been adapted to the new file system; the current strategies both perform reasonably well and are reliable. However, analysis of the Mach/0.5 kernel showed a number of deficiencies in the existing approach. Many of these deficiencies were caused by our self-imposed constraints of providing a significant performance increase in a short period of time and with



minimum modification to the existing code.

In designing a new locking strategy, an important consideration was the performance of the current system. Performance analysis was done by collecting statistics on each lock within the system. These statistics included the number of lock attempts, failed attempts, maximum, minimum and average time the lock was held, etc. This analysis showed that several locks were held for long periods of time, thus reducing parallelism and decreasing performance. While our existing locking strategy may work well, the release of Mach version 2.5 with a vnode-based file system afforded us the opportunity to do better. Unfortunately, time constraints prevented us from addressing every issue.

Mach has two lock primitives from which all of the locking strategies are built. The first type of lock, called a *simple lock*, is implemented as a spin lock. When this type of lock is used, a thread waiting to acquire the lock will spin in a tight loop until it gets the lock. The second type of lock is a blocking lock. It is from the blocking lock that both *read/write locks* and *mutual exclusion (mutex) locks* are derived. With this type of lock a thread may block while holding this lock, waiting for some event such as an I/O completion, to occur. In the case of a mutex lock, only one thread may hold the lock at a time. Read/write locks permit multiple readers to use the lock simultaneously, but only one writer.

Our primary goal was to reduce the time locks were held. When a thread holds a blocking lock for a long period of time, performance is not only degraded by reduced parallelism but also as a result of increased context switches. Performance of portions of the system not related to the data structure the lock is protecting is thus effected as well, since the time spent context switching could be better used computing. The following sections give details of how various data structures were locked in the Unix File System, the Virtual File System and the Network File System.

### 3 Unix File System (UFS)

In our Mach/0.5 release a single mutual exclusion lock protected the inode. This lock was held for long periods of time and prevented simultaneous access to the inode. This strategy was essentially the original Unix inode locking strategy. In many cases, this serialization did not affect performance; access from multiple programs tend to spread across multiple files. However, there are numerous files which are often read frequently, particularly on a large system such as a 20 processor Encore Multimax with hundreds of users. Files such as */etc/passwd* and */etc/group* as well as directories such as */*, */tmp*, */bin* and */usr/bin* are examined frequently. Our performance analysis has shown that system-wide performance could be enhanced significantly by reducing lock contention on such inodes.

To accomplish this, the inode's single mutex lock was divided into two locks; a simple lock to protect the inode contents, and a read/write lock to protect the file across I/O operations. Neither of these locks protect certain fields which are considered to be "read-only". Fields such as the *i\_dev* and *i\_number* are initialized when the inode is created and will not be changed, hence there is no need to serialize access to them.

A simple lock, called the *in-core lock*, guards various pieces of the inode structure. Fields such as *ic\_mode*, *ic\_nlink*, *ic\_uid* and *ic\_gid* are protected by this lock. In all cases, these are fields whose contents are often read but infrequently written. During those cases where the field must be modified, the new value is not dependent upon the result of an I/O operation. A simple lock is sufficient to protect this data, since access time to these variables is brief. Threads needing this data will not have to wait long, and unnecessary context switches are avoided.

A read/write lock, called the *IO lock*, protects both file dependent inode data as well as the

file itself. Unlike our Mach/0.5 implementation, a read/write lock is used to allow simultaneous access to the file's contents when no updates will take place. We have a program that displays the number of times each system call has been called since the system was booted. We typically find a 2:1 ratio of reads to writes on our systems. Since the majority of I/O operations are for reading, a significant performance increase is realized by this change. When applied to directories, the inode read/write lock works as expected. For pathname lookup operations the lock need only be taken for reading. Updating the directory's contents requires taking the lock for writing.

Section 7 discusses the results of our performance analysis. While our results indicate significant performance enhancements as a result of these changes, it is also both important and interesting to examine cases where these changes made little or no impact. Our analysis shows that changing the inode lock from mutual exclusion to read/write had significant impact. There are many inodes that see a large number of simultaneous readers. We have increased parallelism where previously these threads would have blocked. However, there are files whose pattern of usage will not benefit either from splitting the original inode lock into two separate locks nor using a read/write lock. Those files that are typically write-only or append-only files will still have access effectively serialized by the write lock. An example of such a file is `/usr/adm/acct`, which is primarily an append-only file. Providing fine-grained locks, with a different locking scheme, would only benefit environments with significant append-only file access patterns. Fortunately, this is not the typical usage within a general purpose time-sharing system.

We do not take the IO lock for reading and writing to block or character files. Therefore, for both block and character special files only simple locking is performed at the UFS level. We retain the assumption of our original locking scheme, that the buffer cache will synchronize reads and writes to the same file blocks and that the device driver is responsible for character device synchronization.

Another significant change for inode locking was to the inode cache. The inode cache is composed of hash buckets with a linked list attached. Mach/0.5 used a mutual exclusion lock on each hash bucket as well as on the inode free list. A "worst-case" scenario was for `iget` to lock the appropriate inode hash chain; not find the desired inode in the cache; acquire the freelist lock; wait for the I/O to complete; then unlock the two locks. Performance analysis showed that these locks, with the inode freelist lock in particular, had some contention. These locks were replaced with spin locks, which are released across operations that may block. However, the increased parallelism comes at a price.

It is now possible that two threads may simultaneously attempt the lookup of the same on-disk inode. The second may insert the in-core copy of the on-disk inode into the cache before the first thread completes its construction of the in-core inode. To compensate for this, after finding a free inode the thread reacquires the hash-chain lock and searches the chain. If the inode is now present, the thread has lost the race. It frees the new inode it constructed, and uses instead the inode that the other thread obtained for it. Otherwise, it inserts the inode in the chain and proceeds to read the inode from disk. All threads needing the inode then wait for the I/O to complete. This race condition is rare in comparison to the frequency with which the desired inode is found in the cache. Hence, the re-search operation is not a big price to pay as compared to the increased performance gained by reducing lock contention and rescheduling.

## 4 Virtual File System (VFS)

### 4.1 Directory Name Lookup Cache

Sun's Directory Name Lookup Cache (DNLC) improves pathname translation performance by providing fast lookups of frequently used file names. The cache is indexed by a hash value obtained from both the file name and directory vnode. A simple lock was added to protect each of these hash chains, as well as the LRU (Least Recently Used) chain.

As an example of its use, when a name is to be entered into the cache the following steps are taken:

1. Compute hash chain index and acquire that hash chain lock.
2. Search linked list to determine if entry already exists. If so, release hash chain lock and return.
3. Acquire the LRU chain lock.
4. Obtain a free element from the list. To get a free element, the hash chain of which this element is a member must be locked so it can be removed. If we are unable to obtain this lock, we move on and try the next element in the LRU list.
5. Remove entry from LRU chain. The LRU chain is then unlocked.
6. Initialize entry with new information.
7. Re-acquire the LRU lock. Insert entry on the chain and release the LRU lock.
8. The entry is inserted into the appropriate hash chain and the hash chain lock may then be released.

When obtaining the free element from the LRU chain, the thread will not wait on the hash chain of the "oldest" element. This is to avoid deadlock. The deadlock can occur when another thread is holding that hash chain lock, waiting for the first thread to release the LRU lock. Instead, the thread looking for a reclaimable cache entry will just continue to traverse the LRU list until it finds an unused entry or a used entry on an unlocked hash chain. This method may result in an earlier removal of an entry, but, it increases single-stream performance by not forcing the thread to spin waiting on entry's hash chain lock. It also helps us maintain our goals of minimal modification to the code and the existing algorithm.

Note that the LRU lock is acquired twice during this process. Reacquiring locks is not unique to this particular operation. Our general philosophy was that we were willing to make small sacrifices to single stream performance provided that additional parallelism will be gained in a multiprocessor environment by reducing lock contention.

### 4.2 Vnode

To reduce lock contention, in a manner similar to inodes, we divided the vnode fields into three categories. The first contains fields used during lookup operations such as pathname translation. Only one field, *v\_vfsmountedhere*, is currently protected by the lookup lock. The lookup lock is a read/write lock. The second category contains the data fields modified during I/O operations. This in-core lock is a simple lock. The fields it protects include the vnode's flags, its reference count and its shared and exclusive user-level file lock state. All other fields of the vnode are considered

read-only. These are set when the vnode is allocated and never change during the lifetime of the vnode.

The lookup lock is taken for writing during mount and unmount operations. During a mount operation, we make sure there are no new references to the vnode being mounted on by taking the vnode's lookup lock for writing. This effectively disables pathname translation through the vnode. We may then obtain an accurate reference count on the vnode and proceed with the mount. On an unmount operation, we disable lookups on the vnode that is covered by the mounted filesystem immediately before calling `dounmount`. The `dounmount` function has been modified to take the covered vnode pointer rather than the `vfs` structure pointer as a parameter. This modification was necessary because there could be a race between two threads unmounting the same pathname. They would serialize on the lookup lock right before calling `dounmount`. However, the second thread would wake up and use a dangling `vfs` pointer, which has already been properly taken care of by the first thread doing the unmount.

The lookup lock is taken for reading when pathname translation is done. By taking the lock for reading, any threads trying to take the lock for writing are blocked out. This prevents the unmounting of the file system while translating the pathname, but before referencing the vnode. Once the vnode is referenced, the read lock is released and any waiting unmounts will return an error signifying the filesystem is busy. However, by taking the lock for reading, multiple pathname translations may proceed in parallel. Using this scheme, simultaneous pathname translations. no longer serialize while processing common pathname components such as `/`.

The vnode in-core lock will be taken whenever any of the above mentioned fields is modified or accessed. Even though these fields are frequently referenced in some manner, contention on the simple lock has not been a problem. The `VN_HOLD/VN_RELE` operation to reference/dereference a vnode is very common throughout the sources for `vfs`. The count will be incremented or decremented every time we reference or dereference the vnode. A spin lock is sufficient for this because the count does not depend on any information from a blocking operation, and because the increment or decrement is a very short and fast operation. Any thread spinning on the in-core lock will not have to wait very long.

## 5 Network File System

Clearly, NFS is the *de facto* standard today for commercial distributed filesystems in the Unix environment. Given the demands of our local computing environment, and the demands from our customers, we sought NFS performance close to that of Encore's commercial operating systems with as little effort as possible. However, we were unwilling to import code from either of Encore's commercial products because their code was extensively modified and we preferred to start with source code as close to the original as possible. We also wished to avoid additional licensing requirements that would have been imposed by using code from the other operating systems.

Since NFS uses remote procedure calls, we decided to parallelize the code in such a way that each `rpc` could be operated on separately. We specifically did not intend to increase performance by somehow arranging for multiple threads to operate on the same `rpc`. The stateless NFS protocol inherently offers a large degree of parallelism because, in theory, multiple `rpcs` need not be synchronized. In practice, of course, there are a number of shared data structures to be protected, such as caches. Of the three layers in the Sun NFS implementation, the Remote Procedure Call and External Data Representation (*XDR*) layers required the fewest changes while the NFS layer required the most.



## 5.1 External Data Representation

The function of the XDR layer is to translate machine-specific data formats into “network byte order” for outgoing messages and to reverse that process for incoming messages. This layer allows heterogeneous machines to provide NFS services to each other over a network. Because the XDR routines operate on data contained in individual rpcs, theoretically these routines required no modification whatsoever for our parallelization scheme. In fact, nearly all the XDR routines were clean: one routine, `xdr_mbuf`, used mbufs through the appropriate macros and so was “invisibly” parallelized by the changes we had made to the macros. A few routines, particularly some NFS-specific XDRs, required parallelization. Overall, however, the number of changes to the XDR layer was negligible.

## 5.2 Remote Procedure Calls

The *RPC* layer required a number of modifications for correct parallel operation. Some of these changes were needed because the RPC layer makes use of a few existing data structures that already have been parallelized and thus new code must also obey their locking rules. Examples include the hostname variable, credentials, and sockets.

The *hostname* variable is set outside the RPC layer. It is typically set at boot-time and only read after that, but the possibility remains that it could be altered at any time. It is protected by a blocking read/write lock, so, in ordinary use, there is no need to wait to use the variable. All of the RPC routines that used the hostname variable required very small modifications to obey the locking rules.

When composing remote procedure calls, it is necessary to include information identifying the sender. These *credentials* are part of a Mach task’s resources. They include the sender’s real and effective user and group ids. Thus, multiple threads within the task may share the same set of credentials, and one thread may in fact alter the credentials while another thread is using them. There are several other ways for credentials to be shared, so clearly in a multiprocessor environment all credentials users must synchronize their activities. Credentials are used throughout the VFS and UFS layers, as well as within the RPC routines. The RPC routines were modified slightly to obey the same locking rules for credentials structures that the VFS and UFS routines use.

We also had well-defined rules for manipulating *socket* data structures, which required the addition of locking code to existing RPC routines. All of these modifications were straightforward.

We found it necessary to introduce new locking protocols for a number of RPC data structures that were used only within the RPC layer but had global scope. There were a number of variables that were defined globally or statically simply for convenience. While these definitions make sense on a uniprocessor, when only one (non-preemptible) thread at a time can use such global or static variables, on a multiprocessor such variables must be eliminated or appropriately protected. One routine used a static but global credentials structure to hold intermediate results. We used a local credentials structure, instead. Another routine, intended to format error strings, returned a pointer to a static buffer. We had no choice but to change the interface; even if we changed the routine to use dynamically allocated memory, we would have to change its callers to free that memory. The new interface required the caller to supply the buffer to use.

Aside from these “nuisance” cases, there were a number of significant global data structures that required some consideration. There are many statistics data structures that record the frequency of common actions. We have taken a casual approach to these global variables. By default, the kernel employs no locks around these structures but a compile-time switch is provided to enable such locking, if necessary. We devised this option because the time spent locking and unlocking



these structures is large relative to the action taken (*e.g.*, incrementing a counter). Properties of the Multimax caching protocols guarantee that, while a counter may occasionally lose one of a number of simultaneous updates, the resulting value will always be the valid result of one of those updates and will never assume some garbage value produced by colliding updates. (It is precisely because of this cache dependency that we felt it necessary to be able to enable statistics locking for architectures that do not make the same guarantees.) Typically, these “almost accurate” counts are close enough for our needs.

Another global data structure requiring consideration is the service callout table, *svc\_head*. This table is protected by a read/write lock. It is usually taken for reading, except when services are added to or deleted from the table. Since table modification is an infrequent operation, the contention on this lock is very small.

When we were all done with the RPC layer, we had modified only 9 of 29 files. Most of the modifications were very small, requiring changes only to obey already existing locking rules or adding locks to existing data structures and algorithms. None of the modifications required significant alteration of the existing RPC data structures or algorithms.

### 5.3 The Network File System Layer

The NFS layer consists of two major subsystems: a *server* subsystem responsible for processing the requests of other machines for local files and a *client* subsystem that converts requests for files located on other machines into the appropriate remote procedure calls. The server-side implementation has very little state to maintain; it is responsible only for handling remote requests that in theory have no interdependence. In fact, the server-side was easy to parallelize.

There were only three issues to face when parallelizing the NFS server code, two of which were simple. There was the standard issue of synchronizing updates of the NFS statistics variables, which was resolved using the same casual approach discussed in the previous section. There was minor work to be done to tailor the NFS system calls to obey the locking protocols already defined for *file* structures and *sockets*.

The third, and most interesting issue, involved the problem of translating the file handle (*fhandle*) in a client's request to its local representation as a *vnode*. The original algorithm broke this process into two steps: first, find the filesystem (*vfs*) in question, and then find the specific *vnode* on that *vfs* specified by the *fhandle*. In our VFS implementation, however, an unmount request can be processed between the first and second steps of the procedure, possibly causing the second step to use invalid *vfs* information. We devised a special routine that combined both steps to avoid racing unmount.

The client-side of the NFS implementation, on the other hand, required considerably more effort than the server-side to parallelize. In fact, parallelizing the NFS layer itself was in many ways comparable to parallelizing the UFS layer. Many data structures required synchronization. For example, the *mntinfo* structure records information about a mounted NFS filesystem and the *rnode* records information about a particular remote file. The *rnodes* are kept in an *rnode* cache, which uses hash chains and a freelist. For correct multiprocessor operation, we added locks to all of these structures, which required changes throughout the code that manipulates them. Most of these changes were straightforward although, as with the other layers, the interesting problems involved preventing deadlock.

Perhaps the most interesting case arose in one of the *rnode* cache manipulation routines, *rinval*. It traverses the *rnode* cache, looking for *rnodes* that belong to a given *vfs*. Having found a candidate, the appropriate subroutine, *binvalfree*, is invoked to invalidate the buffers belonging to that *rnode*. In the uniprocessor implementation *binvalfree* does not block, so the state of the

rnode cache can't change while rinval is walking the hash chains.

In our multiprocessor implementation, traversing an rnode cache hash chain can only be done while holding that hash chain's lock, which is a spin lock. The state of unlocked hash chains obviously can change regardless of whether the thread calling rinval ever blocks. What's more, our version of the binvalfree call may block on buffer cache locks. So, the rnode cache hash chain lock may not be held across that call. Thus, it is now possible for the state of the rnode cache hash chain to change unexpectedly. (In our implementation, the caller of rinval is responsible for guaranteeing that no *new* rnodes for the filesystem in question are added to the cache while rinval is active.)

One possible solution is to re-start the rinval search loop from the beginning of the hash chain in question, or even from the beginning of the entire cache. Performance considerations aside, neither solution suffices because livelock results. That is, rinval will not terminate because it does not remove rnodes from the cache, so it will always invoke binvalfree on the same rnode, then re-start the search. Rinval must always go forward through the cache. The solution is that the next rnode on the chain must be remembered and its reference count incremented so it can't disappear. With this simple modification, rinval can traverse the rnode cache hash chains without livelocking or using stale rnodes.

## 6 What Didn't Change

While there were significant performance enhancements made during the development of Mach/0.6, there were also a number of areas that were *not* modified. In some instances, we are satisfied with the current level of performance. In others, we simply haven't had the resources to investigate the problems. This section discusses those areas and the reasons why these areas were not enhanced.

### 6.1 Network

No performance enhancements were made to the network. We continue with the goal that we will optimize only those areas that prove to be performance bottlenecks. Performance analysis [2] indicates that our efforts are best spent elsewhere.

### 6.2 Buffer Cache

Protection of the buffer cache consists of a mutual exclusion lock on each of the hash chain headers as well as a mutual exclusion lock on the buffer itself. Statistics indicate that there are a number of worthwhile buffer cache optimizations possible. For example, a read/write lock could be used on the buffer instead of a mutual exclusion lock. Simple locks could have been used in place of the blocking lock protecting the hash chains. While such optimizations were considered, our eventual goal is to remove the buffer cache in its entirety and use the Mach "No Buffer Cache" code instead. Unfortunately, CMU has not yet completed this effort.

### 6.3 Accounting

If accounting is enabled via the *acct* system call, an accounting record will be written to the specified accounting file by each process on process termination. There is a single mutex lock protecting the accounting related data structures. Examining system call counts indicates that

process exit is approximately one tenth of one percent of all system calls. Average wait time on the accounting lock is approximately 125,000 micro-seconds. While the maximum time a thread has waited for this lock exceeds 1.1 seconds, the lock is obtained immediately 96% of the time.

Unix time-sharing systems tend to perform a significantly larger number of process creations and terminations than other systems, like a database environment. Considering this, the 4% miss rate on the accounting lock did not appear to warrant rewriting the accounting portion of the system.

## 6.4 Disk Block Allocation

The disk block allocation algorithms used in Mach/0.5 were not changed in the Mach/0.6 software. There are two protection mechanisms employed when allocating disk blocks. Cylinder groups are fetched through the buffer cache, so the buffer lock implicitly protects the cylinder groups too. A mutual exclusion lock synchronizes access to the filesystem superblocks. The current file system lock is not on a per file system basis, as might be expected, but is actually a global lock. It was our intention to implement a per file system lock, but lock statistics indicated that contention on this lock was so low that no change was warranted.

## 7 Performance

We configured an Encore Multimax 320 system as a test machine with 5 Advanced dual Processors Cards (APCs), each of which has two National Semiconductor 32332 processors rated at 2 MIPS apiece, and 32 megabytes of main memory. Each processor has a small, private 64 kilobyte write-through cache with snoopy logic to maintain cache coherence. We used the same filesystems and build tools on all the tests.

Before analyzing the performance of our parallelization changes, we first had to determine the relative performance of the uniprocessor Mach/0.5 and Mach/0.6 kernels. (In other words, kernels that contained Mach and Berkeley compatibility code, as delivered by Carnegie-Mellon University, without any of the Encore parallelism enhancements.) We assumed that the filesystem performance on both versions of the uniprocessor kernel would be roughly equivalent. To our surprise, a number of tests demonstrated that the newer kernel was actually somewhat slower. For example, we used *NFSstones*, a benchmark devised to analyze NFS performance, to measure local disk performance[6]. *NFSstones* uses a mixture of operations designed to test NFS performance, such as opens, closes, reads and writes in proportions similar to those observed on working systems. *NFSstones* consistently yielded about 130 *NFSstones*/second on the Mach/0.5 kernel but only about 100 *NFSstones*/second on the newer Mach/0.6 kernel. In general, we found that the Mach/0.6 kernel ran roughly 30% slower than the Mach/0.5 kernel on local filesystem tests.

We were not interested in optimizing single-stream performance at the time we ran these tests (although we will do so in the future), so the parallelism results we report for Mach/0.6 are colored by an inherent handicap when compared with Mach/0.5. Overall, however, the parallelism in Mach/0.6 overcomes the single-stream disadvantage it faces when compared with Mach/0.5.

We created a number of tests designed to probe the differences between Mach/0.5 and Mach/0.6 with respect to pathname translation, filesystem parallelism, and lock contention. The pathname translation test repeatedly issued open system calls on a multi-component pathname on a UFS filesystem to test the performance of the vnode translation mechanism, the vnode lookup lock, and the directory name lookup cache. Secondly, of course, we were also testing the inode spin lock.

We used the hardware microsecond counter built in to the Multimax to time the open system

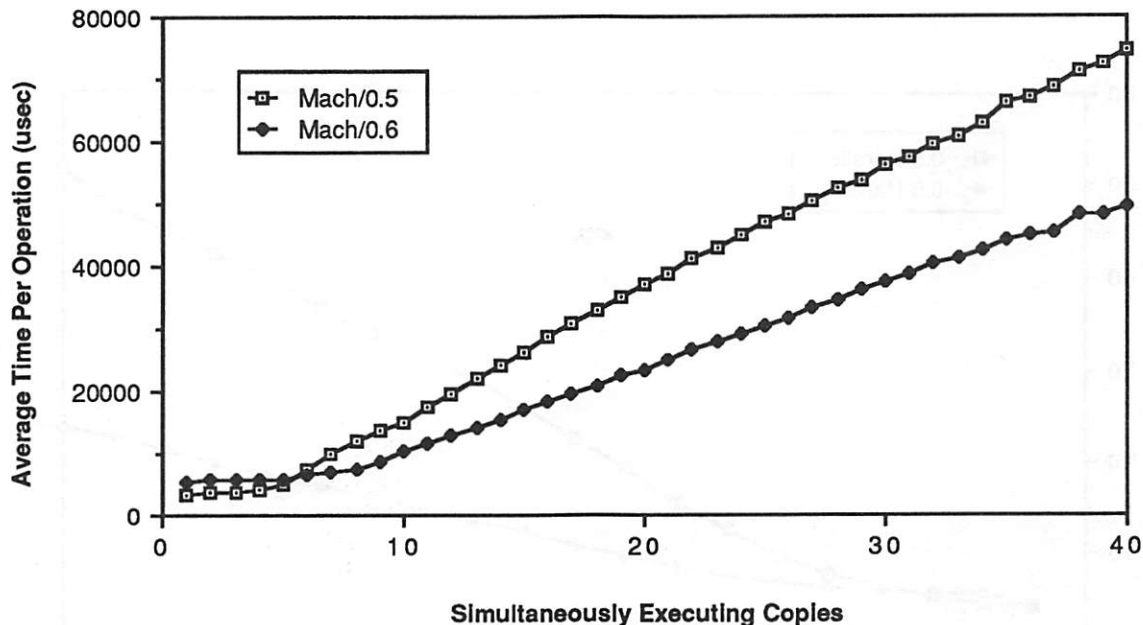


Figure 1: Translating A Single Pathname

call. A test series consisted of first running one copy of the test, then two copies simultaneously, then three, and so on, up to forty. We ran three test series each on Mach/0.5 and Mach/0.6 and averaged the results.

We expected the components of the pathname to be cached in the DNLC; we sought to present significant contention to the vnode layer and the DNLC for the same vnodes, directory name cache entries, and the DNLC's freelist lock. Of course, some work is also required of the UFS layer. We found that initially Mach/0.6 yielded slightly slower results than did Mach/0.5 (see Figure 1). We attribute most of this difference to the fundamental performance difference we described above, although lock overhead may have played a small part as well. However, the additional inherent parallelism in the Mach/0.6 vnode layer quickly becomes apparent.

To compare the inode read/write lock implemented in Mach/0.6 against the inode mutual exclusion lock used in Mach/0.5, we implemented a file read test that repeatedly fetched the contents of a small (50 kilobyte) file, using the microsecond counter to time the read system call. Time to open and close the file did not count, although the test *does* include the time to rewind the file pointer back to the beginning of the file. This rewind time is negligible and affects all copies of the test equally. (Note that each copy of the test separately opens the file, so the tests do not share file descriptors or file structures.) Each test series consisted of executing from one to twenty simultaneous copies of the read test. We ran three test series each against Mach/0.5 and Mach/0.6, and averaged the results. To no one's surprise, the inode read/write lock is shown to be a clear win over the mutual exclusion lock (see Figure 2).

In daily use, we find that sequential reads from small files are common in a Unix-compatible environment, especially on shared system files that are infrequently modified. The ability for users to read simultaneously from shared system files directly affects the ability of the operating system to support large numbers of users. The inode read/write lock also proves useful in situations where users are reading from the same directory or applications are randomly reading from shared files.

We also tested the parallelization of the NFS code. The general results indicate that the NFS parallelization has a substantial impact on performance. We used the same open test for pathname translation we described above to test pathname translation on an NFS filesystem mounted through



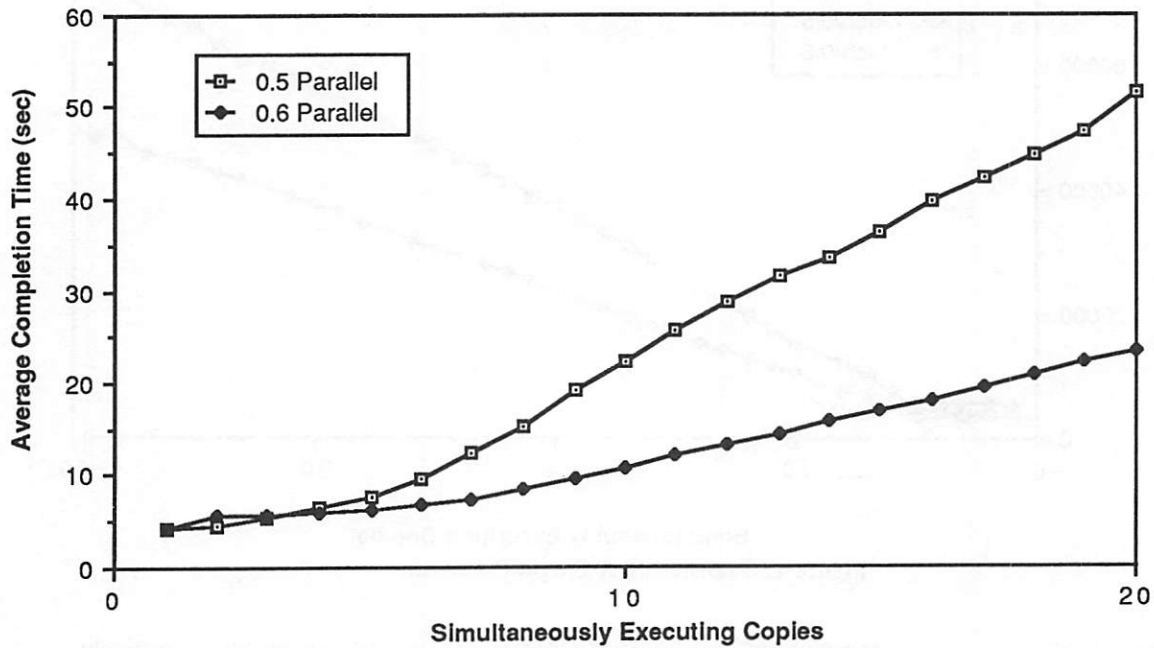


Figure 2: Reading From The Same File

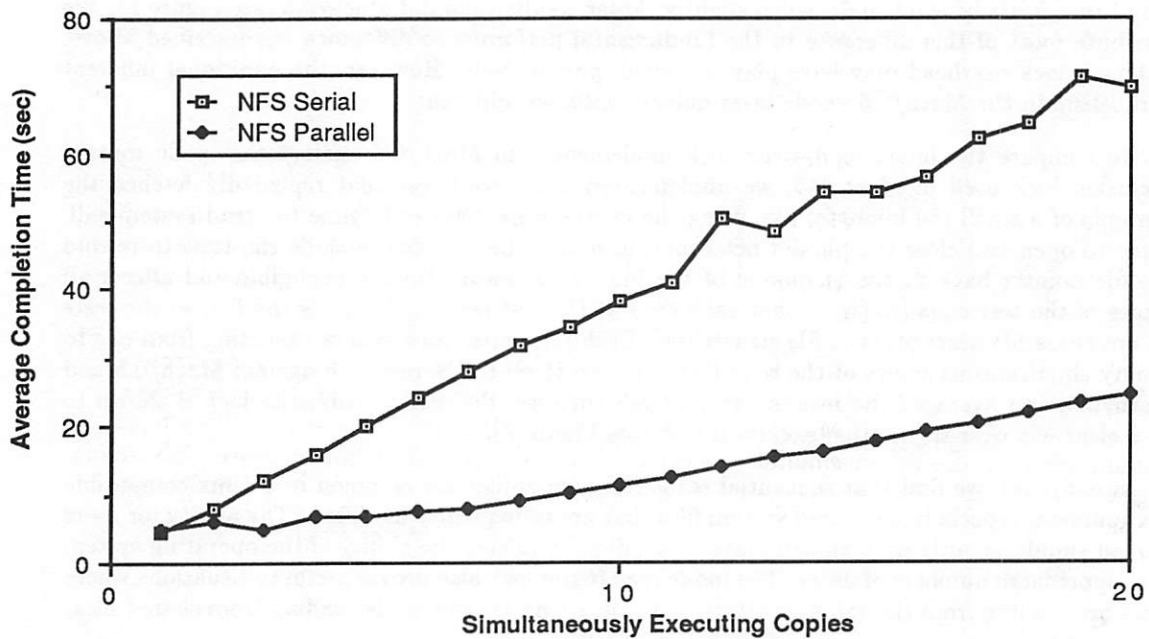


Figure 3: NFS Pathname Translation Test



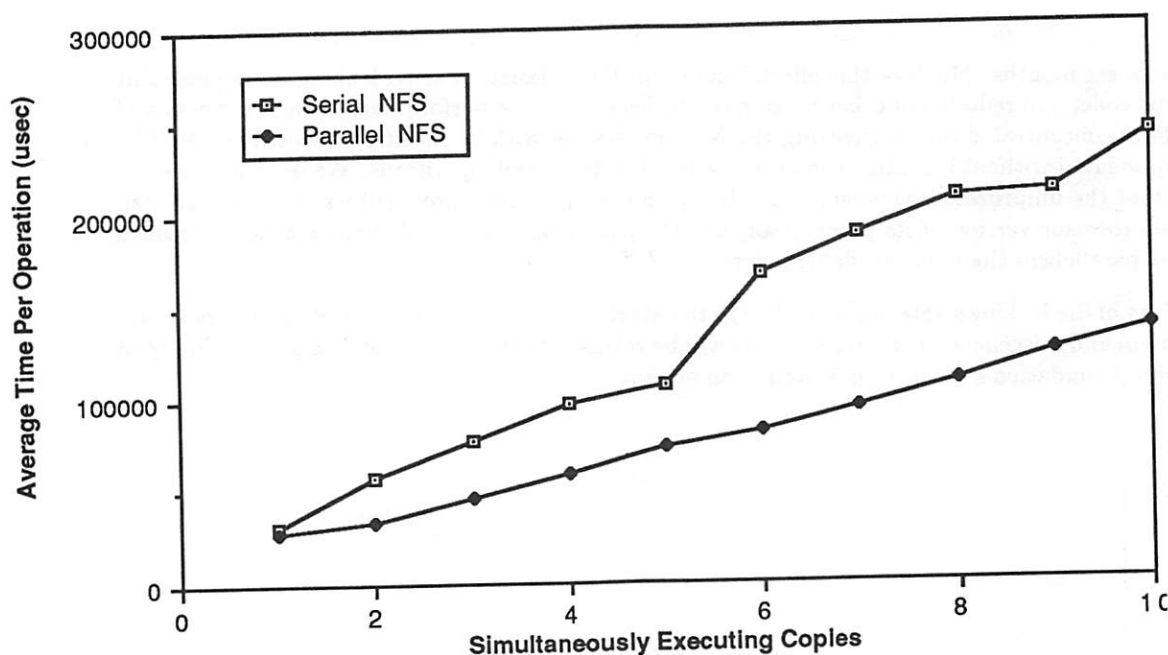


Figure 4: NFS Read Test

the loop-back network interface onto the same host. (We used the loop-back interface because we were not interested in testing out the low-level networking code, just the NFS and UFS code.) While we might expect the results to approximate those of the VFS/UFS pathname test described above, in fact the NFS subsystem has to do more work. In particular, while pathname components cache in the directory name lookup cache, the attributes on rnodes time out, forcing the NFS code to fetch information about the file from the remote system. (In our case, that's the local system as well.)

We ran the tests long enough to be sure that transient effects like attributes timeouts affected all the tests. Each test series consisted of from one to twenty simultaneous copies of the pathname test. We ran three series each on the Mach/0.6 kernel with NFS and our parallelization changes and on the Mach/0.6 kernel with NFS but without any Encore parallelization. As usual, we averaged the results. The results were a little surprising (see Figure 3). There was a substantial performance gain by the parallelized NFS kernel over the unparallelized NFS kernel. We attribute this difference not to optimizations at the VFS layer, like the directory name lookup cache, but to the efficiency of the locking strategies developed for the underlying portions of NFS. This conclusion is based on the larger differences seen with the parallelized VFS/NFS test than with the same test run only through the VFS/UFS layers.

We conclude our test results with a report on rnode read/write locks. In the uniprocessor version of Mach/0.6, lacking any of the Encore parallelism enhancements, rnodes are locked with mutual exclusion locks. Parallelized Mach/0.6, of course, uses read/write locks instead. We constructed test suites of from one to ten simultaneous copies of the file read test described above. The results suggest that the use of read/write locks on rnodes becomes a win as soon as even two threads use the same file (see Figure 4).

## 8 Summary

The integration of the new vnode-based filesystem with the parallelization changes we had developed for the older inode-based filesystem required substantial effort from four people over the

course of six months. Much of this effort focused on the redesign of some locking strategies to fit the new code, and redesign of other locking strategies to increase performance. Another portion of the effort concentrated on parallelizing the NFS subsystem with as much parallelism as could be accommodated without inventing brand-new data structures and algorithms. We were surprised to learn that the uniprocessor version of the Mach/0.6 kernel actually provided less throughput than the uniprocessor version of its predecessor, but the new locking protocols demonstrably provided greater parallelism than was available before.

Many of the locking strategies embodied in the Mach/0.6 Berkeley compatibility code – network, filesystem and miscellaneous routines – soon will be released to the larger world as part of the Open Software Foundation's Operating System Component.

## References

- [1] M. Bach and S. Buroff. Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 63:1733-1749, October 1984.
- [2] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis. In *Workshop Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 105-126, El Toro, CA, 1989. USENIX Association.
- [3] Encore Computer Corporation, Marlborough, MA 01752-3089. *UMAX 4.2 Programmer's Reference Manual*.
- [4] G. Hamilton and D. Code. An Experimental Symmetric Multiprocessor Ultrix Kernel. In *Conference Proceedings, 1988 Winter USENIX Technical Conference*, Berkeley, CA, 1988. USENIX Association.
- [5] R. Rashid. Threads of a New System. *Unix Review*, August 1986.
- [6] B. Shein, M. Callahan, and P. Woodbury. NFSSTONE: A Network File Server Performance Benchmark. In *Conference Proceedings, 1989 Summer USENIX Technical Conference*, pages 269-276, El Toro, CA, 1989. USENIX Association.
- [7] U. Sinkewicz. A Strategy for SMP ULTRIX. In *Conference Proceedings, 1988 Summer USENIX Technical Conference*, pages 203-212, El Toro, CA, 1988. USENIX Association.
- [8] A. Tevanian, Jr., R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. Mach Threads and the Unix Kernel: The Battle for Control. In *Conference Proceedings, 1987 Summer USENIX Technical Conference*, pages 185-198, Berkeley, CA, 1987. USENIX Association.



**Alan Langerman**  
*Encore*

Alan Langerman has been a software engineer at Encore Computer Corporation for the past 3.5 years, working on various Unix dialects and the Mach operating system. He holds most of a bachelor's degree in Computer Science from Harvard University.



**Joseph Boykin**  
*Encore*

Joseph Boykin is employed as a Principal Software Engineer with Encore Computer Corp. In this position his responsibilities include the design and implementation of performance enhancements to the Mach Operating Systems for use on parallel computer architectures.

Prior to joining Encore Computer Mr. Boykin worked as the Senior Partner in a consulting firm and as a Project Leader at Data General Corp. When at Data General Mr. Boykin led a group designing and implementing a multi-processor 68000 based UNIX workstation running System V UNIX.

Mr. Boykin has held several leadership positions within the IEEE Computer Society including chairing the Technical Committee on Operating Systems; Vice-Chair of Technical Activities; Treasurer and Chair of the Workshop on Workstation Operating Systems. Mr. Boykin is currently guest-editor for Computer Magazine preparing a

special issue on Operating Systems for May, 1990.

Mr. Boykin holds both an M.S. in Computer Science and an M.A. in Psychology. His graduate work was done at the Ohio State and Pennsylvania State universities.



**Susan LoVerso**  
*Encore*

Susan LoVerso has been a software engineer at Encore Computer Corporation for the past 2.5 years, as a member of the Mach Operating Systems development group. Prior to joining Encore, she received her Masters degree in Computer Science from the State University of New York at Buffalo.

Sue is a member of the IEEE Computer Society and has contributed significantly to the preparation of the special issue of Computer Magazine on Operating Systems for May 1990.



**Shashi Mangalat**  
*Encore*

Shashi Mangalat has been working with the Encore Computer Corporation for the past year and a half, of which the last 6 months with the Mach Development group. He received his B.S. degree in Computer Science from Oklahoma University and his Bachelor in Physics from Calicut University, India. Shashi has been a member of the ACM and the IEEE Computer Society since 1987.

# Disk Scheduling Revisited

*Margo Seltzer, Peter Chen, John Ousterhout*

*Computer Science Division  
Department of Electrical Engineering and Computer Science  
University of California  
Berkeley, CA 94720*

## ABSTRACT

Since the invention of the movable head disk, people have improved I/O performance by intelligent scheduling of disk accesses. We have applied these techniques to systems with large memories and potentially long disk queues. By viewing the entire buffer cache as a write buffer, we can improve disk bandwidth utilization by applying some traditional disk scheduling techniques. We have analyzed these techniques, which attempt to optimize head movement and guarantee fairness in response time, in the presence of long disk queues. We then propose two algorithms which take rotational latency into account, achieving disk bandwidth utilizations of nearly four times a simple first come first serve algorithm. One of these two algorithms, a weighted shortest total time first, is particularly applicable to a file server environment because it guarantees that all requests get to disk within a specified time window.

## 1. Introduction

Present day magnetic disks are capable of providing I/O bandwidth on the order of two to three megabytes per second, yet a great deal of this bandwidth is lost during the time required to position the head over the requested sector. This study focuses on improving the effective throughput by using rotation and seek optimizing algorithms to schedule disk writes.

Since the introduction of the movable head disk, many people have undertaken similar efforts. However, most of these studies have assumed short queue lengths, and the performance improvement obtained under the various techniques is not substantial. Our approach was to consider a system, such as a file server, with a large main memory dedicated to disk buffering. We assumed that newly written data need not be transmitted to disk immediately; instead, it may be retained for a short period of time in a main memory buffer and transmitted to disk at a time that maximizes disk throughput. Given the ever increasing sizes of main memory (up to one hundred megabytes or more on some file servers), hundreds or thousands of blocks could be queued for writing at any given time. By careful ordering of these requests, it should be possible to reduce average head positioning time substantially. On the other hand, the potential for starvation of a request becomes more important and fairness becomes a requirement. To this end, we have developed two algorithms that attempt to avoid starvation yet provide very good disk utilization.

## 2. Previous Work

Most previous work has dealt with scheduling a small number (fewer than 50) of I/O requests. With small numbers of requests, research concentrated on first come first serve (FCFS), shortest seek time first (SSF), and the scanning algorithms which service requests in cylinder order scanning from one edge of the disk to the other.



Hofri shows that under nearly all loading conditions, SSF results in shorter mean waiting times than FCFS [HOFR80]. The main drawback he finds to SSF is the larger variance in I/O response time. He also alludes to more optimal scheduling which takes into account the number of requests in a given cylinder, but does not pursue this further. Hofri's results are a combination of theoretical analysis and simulation.

Coffman, Klimko, and Ryan also discuss FCFS and SSF [COFF72]. They add to their analysis two scheduling policies which are intended to control the high variance of SSF. These are called SCAN and FSCAN. SCAN restricts its search for the minimum seek time request to one direction (inward or outward). However, SCAN still causes long waiting times for requests on the extremes of the disk. FSCAN addresses this by "freezing" the queue once the scan starts--requests that arrive after the scan starts are serviced in the next scan. By pure theoretical analysis, Coffman et al. concludes that SCAN uniformly results in lower average response times than either FCFS or FSCAN, but higher average response times than SSF. Geist describes a continuum of algorithms from SSF to SCAN differing only in the importance attached to maintaining the current scanning direction [GEIS87].

In [TEOR72], FCFS, SSF, and SCAN are again analyzed. Similar conclusions are made that SSF yields shorter response times than SCAN, which yields shorter response times than FCFS. The Eschenbach scheme, which is similar to SCAN, schedules according to rotational position in addition to seek position. As a result, the Eschenbach scheme generates lower average response than any previous scheme as the queue length increases.

In all of these papers, no queue lengths averaging more than 50 are studied. This limitation is due in large part to the smaller memory sizes of the time and slower CPU's. Now, with exponentially growing memory sizes [MOOR65] and faster CPU's, more data may accumulate more quickly, and disk queues are no longer constrained to small lengths. With large queues, we are able to investigate previously impractical or unnecessary schemes. In particular, we continue the study of rotationally optimal scheduling algorithms.

### 3. The Test Environment

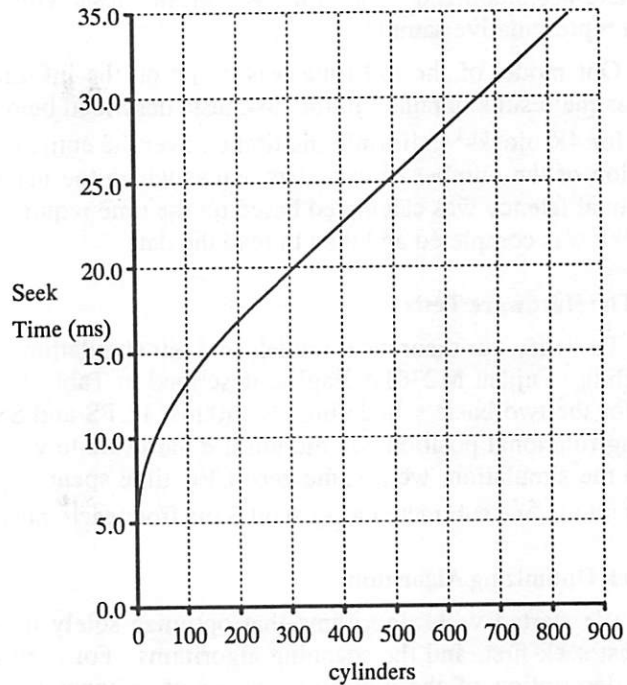
We chose to analyze the algorithms in three ways: theoretical model, simulation, and hardware tests. The theoretical model served as a first approximation of the potential performance gain. The simulation provided the most flexible testing platform, and the hardware tests verified the correctness of the simulator. After the validation of the simulator on some of the simpler algorithms, the remaining results were all derived from simulation.

#### 3.1. The Simulator

The simulator modeled a Fujitsu M2361A Eagle described in Figure 1 and Table 1. In all the simulations, the CPU time required to calculate the next request was ignored on the basis that this computation could be overlapped with the actual I/O operation. Furthermore, we wished to focus on the potential of the algorithms themselves rather than optimizing their implementation.

Since we were most interested in viewing the behavior of the algorithms in the presence of many requests, we introduced an artificial model of request arrival. In order to examine behavior for a queue length of  $Q$ , we initialized the queue to contain  $Q$  events, each with a request time of 0. Whenever a request was serviced, it was replaced with a new request whose request time was equal to the completion time of the completed request. In this manner, we guaranteed that we always had a queue of length  $Q$  from which to select a request and our simulations were insensitive to the real arrival rate. In order to avoid skewing the response time results (by leaving unserved requests in the queue at simulation completion), we completed the simulation by emptying the entire queue. That is, for the last  $Q$  requests, we did not generate any new requests, but

Fujitsu Eagle disk drive	
cylinders/disk	840
tracks/cylinder	20
sectors/track	67
bytes/sector	512 B
average seek	18 ms
average rotational latency	8.3 ms
time to transfer 4 KB	2 ms



**Table 1: Specifications of Fujitsu disk drives.**

**Figure 1: Seek Time Calculation** Graphed above is seek time in ms as a function of seek distance in cylinders [FUJI84]. We model this as

$$seektime(x) = \begin{cases} 0 & \text{if } x=0 \\ 4.6 \text{ ms} + .87 \text{ ms} \sqrt{x} & \text{if } x \leq 239 \\ 18 \text{ ms} + .028 \text{ ms} (x - 239) & \text{if } x > 239 \end{cases}$$

served those remaining in the queue.

In order to guarantee that the averages obtained were statistically significant, we needed to determine an acceptable length for the simulation runs. Let  $Q$  be the length of the queue and  $B$  be the total number of blocks on the disk. At each point in time, there are  $Q$  objects selected from a set of  $B$ , in the queue. Therefore the probability of any particular set of  $Q$  objects being present is  $\frac{1}{\binom{B}{Q}}$ . For the drives we tested, there were 140,280 blocks on the disk. So, for a queue

length of 10, there are on the order of  $10^{44}$  combinations and for a queue of length 1000, there are more than  $10^{100}$  combinations. Clearly, it is infeasible to actually examine a large portion of this space. Furthermore, in our simulations, each sample of size  $Q$  is not independent since the entries in the queue at time  $t$  differ from the  $Q$  entries in the queue at time  $t-1$  by precisely one event.

We define a test run as one simulation which generated a full queue of random I/O requests and serviced them. We analyzed the variance across  $N$  runs where  $N$  ranged from 2 to 200. After 100 runs of size  $Q$ , the variance had decreased to a small fraction (1-2%) of the mean response time and had stabilized. Therefore, we felt that tests with 100 times the number of queued items was a representative sample.

Our model of the I/O time was based on the information provided by the disk vendor as well as the results obtained in the disk tests described below. All requests to the disk subsystem were for 4K blocks<sup>1</sup> uniformly distributed over the entire disk. The seek time was computed as a function of the number of cylinders across which the head needed to move (see Figure 1). The rotational latency was calculated based on the time required to bring the data under the head once the seek was completed and then to read the data.

### 3.2. The Hardware Tests

To verify our theoretical models and our simulations, we ran tests on the disks that we were modeling, Fujitsu M2361A Eagles, described in Table 1 and Figure 1. We verified the simulation for the two basic scheduling algorithms: FCFS and SSF. Because of the difficulty in determining rotational position, we did not use hardware to verify the simulation for other algorithms. As in the simulation, we assume zero CPU time spent to process the I/O. In order to factor out the CPU time, we subtracted a constant 3 ms from each individual disk access<sup>2</sup>.

## 4. Seek Optimizing Algorithms

We started with algorithms that optimize solely on seek distances: first come first serve, shortest seek first, and the scanning algorithms. For each algorithm we evaluated, we provide a brief description of the algorithm, an intuitive theoretical estimate of its performance (where feasible), the actual results, and a graph comparing the simulated versus theoretical results. Our metric for evaluating the algorithms was disk utilization, which we define as the fraction of time that the disk spends transferring data.

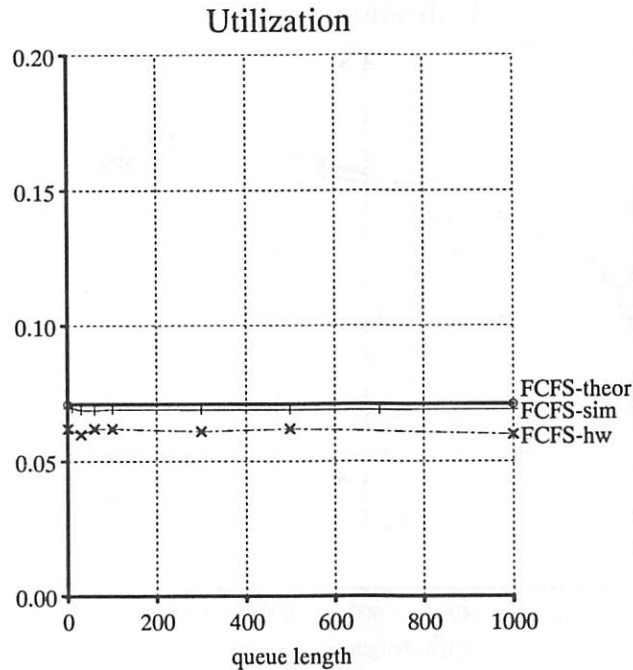
### 4.1. First Come First Serve

The simplest scheduling algorithm imaginable is first come first serve (FCFS). As one would expect, this model is independent of the queue length and we obtain an average I/O time equal to the predicted average seek plus the predicted average rotation. We also used these numbers to verify the other algorithms since any of the algorithms with queue length of 1 should equal FCFS.

The disks were spinning at 3600 RPM yielding a revolution time of 16.67 ms for 67 sectors of 512 bytes each, providing a transfer rate of 4K / 2.0 ms. A back-of-the-envelope calculation will show that for simple first come first serve scheduling policies, we can expect the average I/O time to be one half a rotation (8.3 ms) plus the time for an average seek (18 ms). This yields a disk utilization (the fraction of time the disks are actually transferring data) of  $\frac{2}{2+8.3+18} = 7\%$ . As Figure 2 shows, this is very close to the hardware-derived and simulated utilizations.

<sup>1</sup>We chose a 4 KB block size as a common file system block size.

<sup>2</sup>We estimated the CPU time used in issuing an I/O from a user process by issuing two consecutive I/O's for two sectors on the same track. We found that after reading sector 0, the next sector we could read without missing an entire revolution was sector 12, 1/6 of a revolution later. This implied a 3 ms CPU turnaround time.



**Figure 2:** Comparing FCFS utilization derived from theoretical analysis (FCFS-theor), hardware measurements (FCFS-hw), and simulation (FCFS-sim).

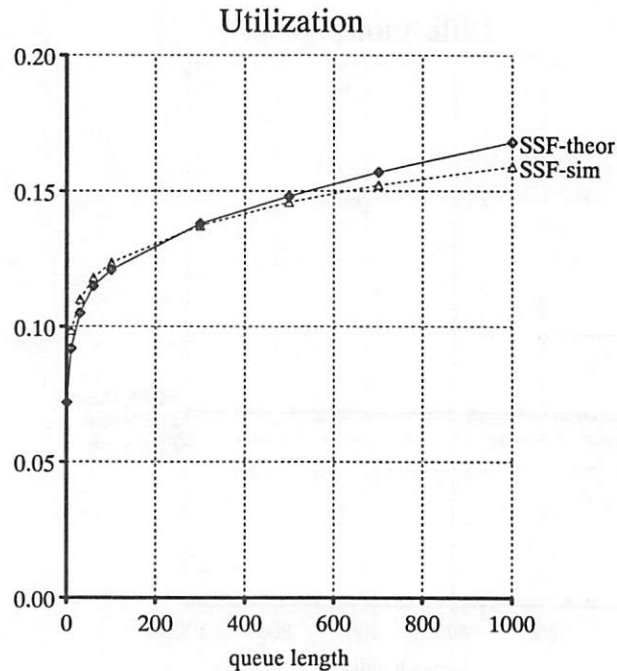
#### 4.2. Shortest Seek First

For this algorithm, we ignore the rotational latency and select requests based on the seek time required. On the average, we expect to see a half rotation (8.3 ms) and a seek to the closest cylinder with a request on it. This is a function of the total number of cylinders on the disk and the length of the queue. For example, the Eagles have 840 cylinders. With a queue length of 100, we expect the average seek to be 8.3 cylinders. Thus, for a queue length of 100, we expect average I/O time to be approximately  $8.3 + 4.6 + .87 * \sqrt{8.3}$  or 15.4 ms.<sup>3</sup> Figure 3 depicts these predicted values against the results actually obtained in simulation. The maximum queue length we used, 1000, corresponds to about 4 MB of dirty blocks. For example, consider a file server with 64 MB of main memory. It is reasonable to assume that 50% may be dedicated to a file cache, and of that, approximately 10-15% (3-5 MB) might be dirty.

#### 4.3. SCAN and CSCAN

The SCAN scheduling algorithm is oriented towards producing fairer response time. It orders the requests by cylinder number and services all the requests for a given cylinder before moving the head to the next cylinder. When the head reaches one end of the disk, it merely reverses direction and begins scanning towards the other end of the disk. It is important to notice that this

<sup>3</sup>This is not quite accurate, as we have used the time of an average seek, which is not the same as the average time. It is a close approximation because of the almost linear seek profile (Figure 1).



**Figure 3:** Comparing SSF utilization derived from theoretical analysis (SSF-theor) and our simulator (SSF-sim).

is actually very similar to the shortest seek first algorithm and we expect similar results.

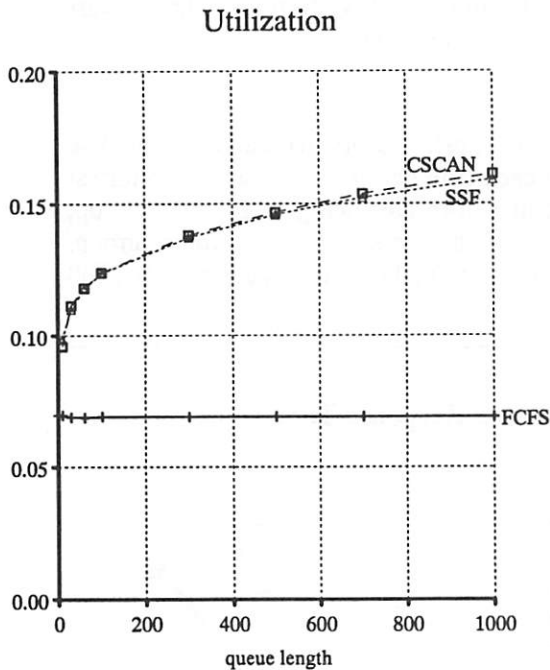
One shortcoming of the SCAN algorithm is that requests on either end of the disks experience worse response time than those in the middle of the disk since those in the middle experience two passes of the head evenly spaced in time whereas the outermost cylinders delay for two full sweeps of all the cylinders before being revisited. Cyclical scan (CSCAN) alleviates this by paying one large seek at the end of the disk to move the head all the way to the other end. That is, the head always moves in one direction and we pay one very long seek at the end of each pass. This long seek is amortized over the requests, and the utilization is nearly the same as SCAN. The major difference is in the maximum observed response times and the variance of the response times. Figure 4 shows the utilizations for CSCAN which are essentially identical to SSF. However, as shown in Figure 5, CSCAN substantially improves the maximum observed response time.

## 5. Seek and Rotation Optimizing Algorithms

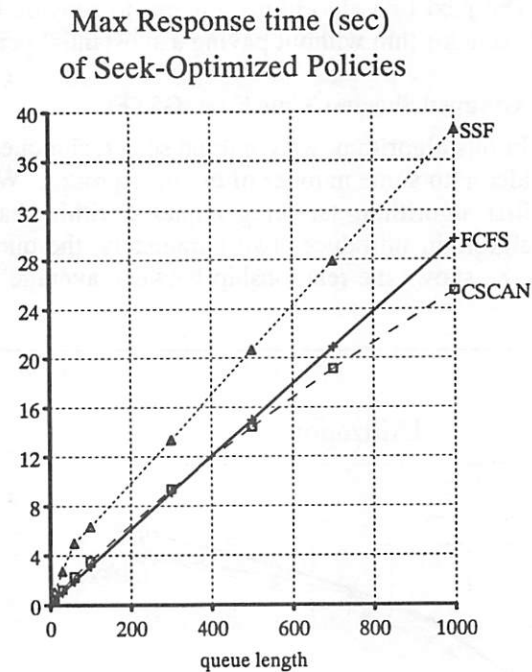
### 5.1. Shortest Time First

In SSF, we chose the request which yielded the fastest seek. In shortest time first (STF), we choose the request which yields the shortest I/O time, including both the seek time and the rotational latency. Advances in disk technology have reduced seek time more than rotational latency. As this trend continues, we expect rotational latency to account for a greater fraction of the total





**Figure 4: Disk Utilization from FCFS, SSF, and CSCAN.** We graph the disk utilization derived from theoretical modeling, hardware verification, and simulation for three simple scheduling policies. The SCAN algorithm yields utilizations almost identical to CSCAN and is not shown.



**Figure 5: Maximum Observed Response Time for FCFS, SSF, and CSCAN.** SSF has significantly worse maximum response time than FCFS, but CSCAN has roughly the same, or lower, maximum response time than FCFS. In some instances, CSCAN is able to have lower response time than FCFS because its average response time is lower.

I/O time, and rotation optimizing algorithms such as STF will become increasingly important. For example, saving half a rotation (8 ms) may cause an access 100 cylinders away to have a shorter total I/O time than an access 1 cylinder away.

STF is expected to yield the best throughput since we always select the fastest I/O. The algorithm scans the entire queue calculating how much time each request will take. It then selects that request with the shortest expected service time. For very long queue lengths ( $Q$  much greater than the number of cylinders), we expect to see the STF time approach 2.0 ms (the time to read a single 4K block). For very short queue lengths, we expect STF to approximate SSF since it is unlikely to have multiple requests on the same cylinder and adjacent requests are likely to be far enough apart so that seek time dominates rotation. Figure 6 shows the simulated results for STF. Note that, even at queue length of 1000, STF utilization is still rising. Preliminary runs at queue lengths of 5000 have utilizations of 40%.

Unfortunately, the scheduling algorithm is a function of both cylinder and rotational position, thus this algorithm is one of the most costly in terms of CPU utilization. In addition, STF

has the potential to starve requests, producing very bad response time (Figure 7). Note that, because the maximum observed response times in Figure 7 were empirically determined, maximum response times in a real system could be even worse.

The next two algorithms attempt to provide the utilization benefits offered by the shortest time first algorithm without paying a substantial penalty in response time.

#### 5.1.1. Grouped Shortest Time First (GSTF)

In this algorithm, we combine scan techniques with shortest time first techniques. The disk is divided into some number of cylinder groups. Within each cylinder group, we apply a shortest time first algorithm, servicing requests within that group before advancing to the next group. This algorithm introduces two parameters, the queue length and the size of the cylinder group. Figure 8a shows the relationship between average I/O times as one holds the queue size at 1000

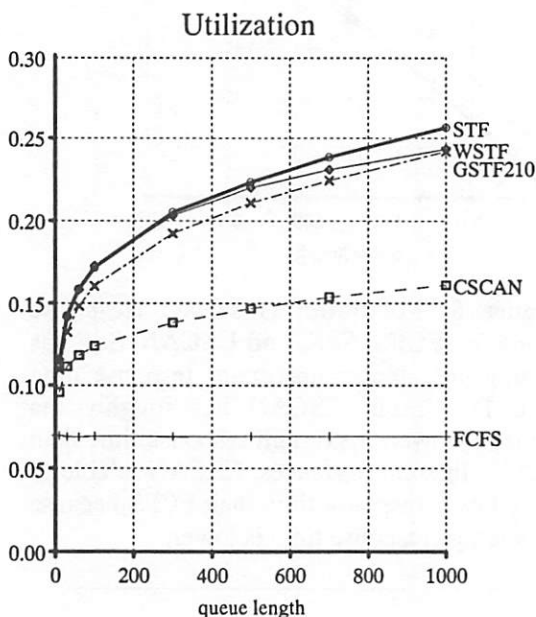


Figure 6: Disk Utilization for FCFS, CSCAN, STF, GSTF210, and WSTF. We graph the disk utilization for seek and rotation optimizing algorithms. FCFS and CSCAN are shown for comparison.

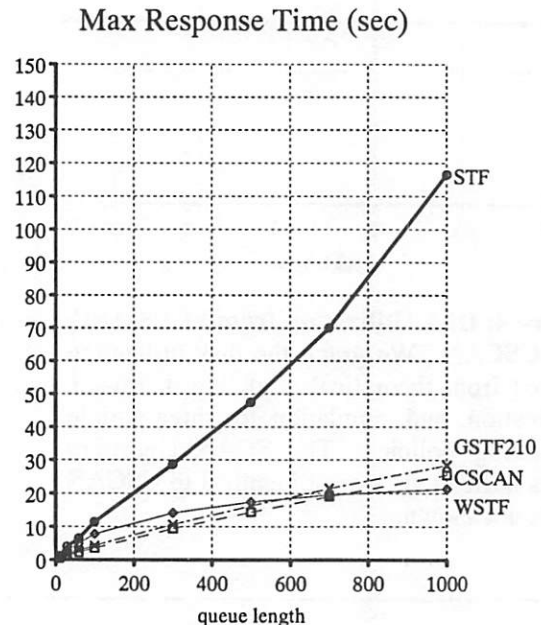
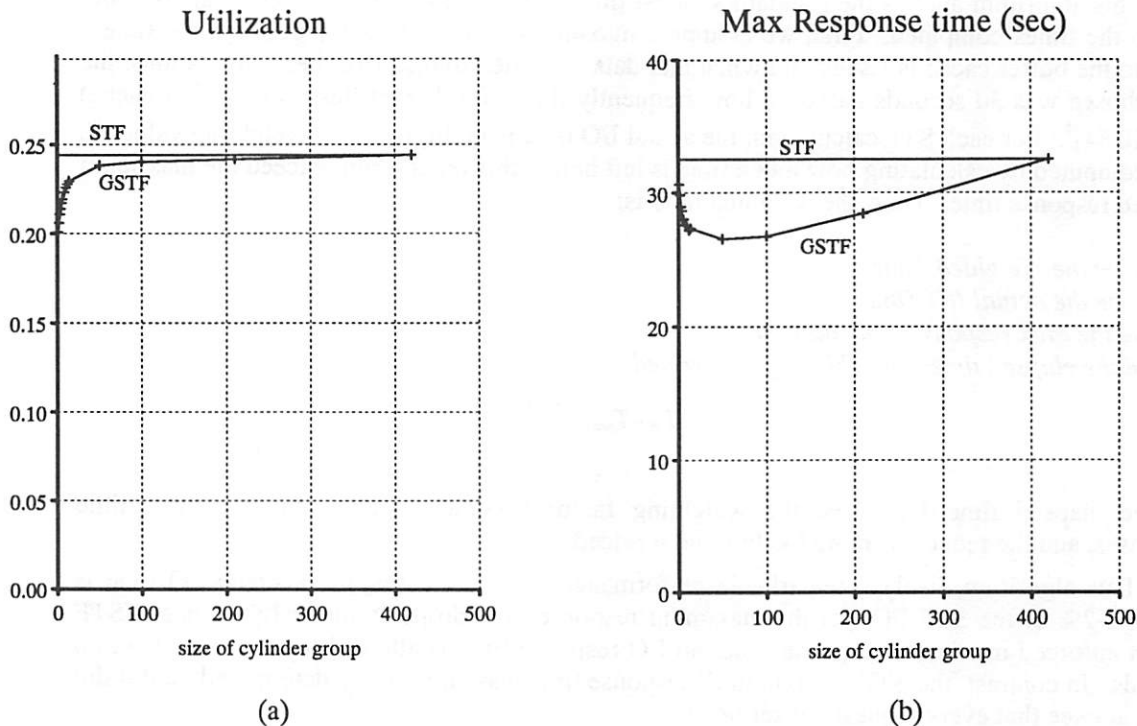


Figure 7: Maximum Response Time for CSCAN, STF, GSTF210, and WSTF. We graph the maximum response time for seek and rotation optimizing algorithms. Maximum response time for STF is much worse than other algorithms, but GSTF210 and WSTF bring maximum response time back down. Recall from Figure 5 that the maximum response time of FCFS is very close to that of CSCAN.



**Figure 8: Utilization and Maximum Response Time of Shortest Seek First and Grouped Shortest Time First.** Graphed above are the utilization and maximum response times as a function of the cylinder group size. The above results are for queue lengths of 1000.

and changes the cylinder group size. Figure 8b shows the maximum response times.

As the cylinder group size increases, the utilization of GSTF increases. Eventually, when the group size is 840 (the entire disk), GSTF is the same (by definition) as STF. Also, as the group size increases, the maximum response time becomes longer and longer, approaching STF. The early dip in the maximum response time curve is due to the interaction of utilization (average disk I/O time) and fairness. Although GSTF is more fair (less variation between response times) at small group sizes, the disk is being used less efficiently, and the average response time is larger. As utilization flattens out, the decreasing fairness causes increasing maximum response times. In Figures 6 and 7, we show the utilization and maximum response time for group size of 210 (4 cylinder groups per disk). We see that GSTF has utilization close to STF, but also succeeds in lowering the response time to close to the maximum response time of CSCAN.

GSTF services all requests for the current cylinder group before moving to the next cylinder group. If requests for the current cylinder group saturate the I/O system, it is possible to starve requests on other parts of the disk. A slight variation of GSTF freezes the queue of a cylinder group as soon as any requests to that cylinder group are serviced, guaranteeing that all the requests within that cylinder group are serviced before the head moves to the next cylinder group. Runs using this variation have 3%-4% lower disk utilizations and 15%-25% lower maximum

response times than the GSTF depicted in Figure 6.

### 5.1.2. Weighted Shortest Time First

This algorithm applies the standard shortest time first technique, but applies an aging function to the times computed. First, we assume a maximum acceptable delay between the time a write to the buffer cache is issued and when that data is written to disk (for these simulations, the time chosen was 30 seconds based on how frequently the UNIX kernel flushes its buffer cache) [MCKU84]<sup>4</sup>. For each STF calculation, the actual I/O time is multiplied by a weighting value  $W$ .  $W$  is computed by calculating how much time is left before this request will exceed the maximum allowed response time. Thus, the weighted time is:

Let  $T_w$  be the Weighted Time

$T_{real}$  be the actual I/O Time

$M$  be the Max response time allowed

$E$  be the elapsed time since this request arrived

$$T_w = T_{real} \frac{M - E}{M}$$

As the elapsed time increases, the weighting factor becomes smaller, the weighted time decreases, and the request is more likely to be serviced.

This algorithm displays remarkable performance. In most cases, the average I/O time is within 1-2% of the STF I/O, yet the maximum response time drops dramatically. Since WSTF has an enforced maximum response time, no I/O response time is allowed to take more than 30 seconds. In contrast, the STF "maximum" response time was empirically determined, and it did not guarantee that every request got serviced.

In trying to understand why WSTF performs so well, it is useful to observe that STF is a greedy algorithm. Always selecting the shortest time first means that regions of the disk get serviced first. However, as regions get cleaned off, there are fewer close requests to service. With WSTF, periodically, the arm is forced to do a "bad" seek, that is, one more costly than another. As a result, the head is in a new region providing the algorithm a better choice of requests from which to select. "Bad" seeks may also occur when a read or a forced write (i.e. a write that must go immediately to disk) is issued. Our results imply that these long seeks are unlikely to harm overall utilization.

## 6. Conclusion

There are two main conclusions from our work. First, substantial performance improvements (on the order of 3 to 4 times) can be gained by these scheduling mechanisms. As the queue from which one selects requests becomes larger, even more improvement can be realized. Second, there are algorithms which achieve this improved performance and still ensure fairness. However, note that at queue lengths of up to 1000, the best algorithms yield less than 40% disk utilization. This still leaves much room for improvement.

The implication is that greater utilization of disk bandwidth is achievable by viewing most of main memory as a large write buffer. In systems where the order of writes is unconstrained, one can take advantage of this unordered nature of writes to minimize the disk seek overhead. Therefore, larger file caches may be used not only to minimize I/O's but to make the necessary I/O's individually cheaper.

<sup>4</sup>One could also adjust the acceptable response time based on the number of requests in the queue (e.g. 150% \* average I/O time of FCFS \* queue length). Runs with these response time limits yielded results within a few percent of runs with the 30 second limit.

## 7. Acknowledgements

We would like to thank Brent Welch and Keith Bostic for instrumenting Sprite and BSD machines to collect statistics on real queue lengths. These numbers convinced us that the burstiness one expects on a file server does, in fact, result in a large number of disk writes to be scheduled.

## 8. References

- [COFF72] Coffman, E. G., Klimko, L. A., and Ryan, B., "Analysis of Scanning Policies for Reducing Disk Seek Times", *SIAM Journal of Computing*, September 1972, Vol 1. No 3.
- [FUJI84] M2361A Mini-Disk Drive Engineering Specifications, Fujitsu Limited, 1984.
- [GEIS87] Geist, Robert, and Daniel, Stephen, "A Continuum of Disk Scheduling Algorithms", *ACM Transactions on Computer Systems*, February 1987, Vol 5. No. 1.
- [GOTL77] Gotlieb, C. C. and MacEwen, H., "Performance of Movable-Head Disk Storage Devices", *Journal of the ACM*, October 1983, Vol 20. No. 4.
- [HOFR80] Hofri, Micha, "Disk Scheduling: FCFS vs SSTF Revisited", *Communications of the ACM*, November 1980, Vol 23, No. 11.
- [MCKU84] Marshall Kirk McKusick, William Joy, Sam Leffler, and R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.
- [MOOR75] G. Moore, "Progress in Digital Integrated Electronics," *Proceedings IEEE Digital Integrated Electronic Device Meeting*, 1975, p. 11.
- [ONEY75] Oney, Walter C., "Queuing Analysis of the Scan Policy for Moving-Head Disks", *Journal of the ACM*, July 1975, Vol 22. No. 3.
- [SMIT81] Smith, A. J., "Input/Output Optimization and Disk Architectures: A Survey", *Performance and Evaluation I* (1981), pp. 104-117.
- [TEOR72] Teorey, Toby J. and Pinkerton, Tad B., "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, March 1972, Vol 15. No. 3.
- [WILH76] Wilhelm, Neil C., "An Anomaly in Disk Scheduling: A Comparison of FCFS and SSTF Seek Scheduling Using an Empirical Model for Disk Accesses", *Communications of the ACM*, January 1976, Volume 9, No. 1.

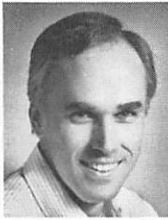
Margo I. Seltzer  
UC Berkeley

Peter M. Chen  
UC Berkeley

Margo I. Seltzer is a Ph.D. student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Her research interests include file systems, databases, and transaction processing systems. She spent several years working at startup companies designing and implementing file systems for transaction processing and designing a full custom VLSI CPU for a multiprocessor architecture. Ms. Seltzer received her AB in Applied Mathematics from Harvard/Radcliffe College in 1983.

Peter M. Chen graduated from the Penn State University (go Lions!) in 1987 with a BS in Electrical Engineering. He received his MS in Computer Science from the University of California at Berkeley in 1989 and is currently a Ph.D. student working with the RAID (Redundant Arrays of Inexpensive Disks) project. His interests are in I/O system design and performance evaluation.





**John Ousterhout**  
*UC Berkeley*

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He and his students have developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is now leading the development of Sprite, a network operating system for high-performance workstations. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.

## Postloading for Fun and Profit

S. C. Johnson

Stardent Computer Corp.  
880 W. Maude Ave.  
Sunnyvale, CA, 94086  
*uunet!ardent!scj*

### ABSTRACT

Postload processing, or *postloading*, is a technique of optimizing (or otherwise processing) an executable program after it has been linked with *ld*. The executable program is read, altered, and rewritten with optimizations or other added features. Postloading, to be done reliably, needs changes to the *a.out* format that increase its size by a few percent. We have found over a dozen significant uses for postloading, including optimization, hardware workarounds, profiling and execution counting, simulation, 'code critics' (programs that examine, and repair, common compiler and assembly code problems), and migration to new hardware releases.

### Introduction

Most code optimization techniques operate locally on an expression, loop, or function. *Postloading*, as the name suggests, is an optimization technique that works after a program has been bound into an *a.out* file.

As an example of a transformation that pays off on many RISC machines, consider accesses to global data. In most RISC architectures, it takes two instructions to read or write a general 32-bit address. In practice, most global accesses happen within a much smaller range, addressable in one instruction as an offset from a 'global pointer.' Assuming a register can be dedicated for use as a global pointer, *ld* would compute offsets from the global pointer as one of its relocation modes. The MIPS execution environment, for one, reserves a register for this purpose, and supports its use in the loader [CHOW86]. (Note that older architectures have similar optimization potential [SZYM78], typically involving replacing longer forms of instructions by shorter forms rather than two instructions by one.)

This optimization is awkward to reflect in the compiler, however. When a reference to a global variable (such as *errno*) is seen, the compiler has to guess whether the variable will lie within the 'fast' single instruction range or whether a 'slow' but general two-instruction sequence must be used. If the compiler always assumed such references were fast, a sufficiently large program would cause loading to fail totally, since there could be too many variables that must be referenced with too few offset bits. Some form of user control over the scope of this optimization seems necessary. However, it is even difficult for the user to know at compile time all the contexts where a program will be used (consider a large set of library functions, such as the X library, for example). If the compiler is pessimistic, much of the benefit of the optimization can be lost.

We propose an alternative. Let the compiler be pessimistic, and produce a fully general, slow sequence. When a program has been linked, then optimize the references in the *a.out* file, replacing two-instruction sequences by a single instruction wherever the target can be so addressed. This strategy allows arbitrarily large programs to be loaded, and for a given loading of the program gives a nearly perfect application of the optimization. The costs are an extra loading step and a few percent increase in the size of the *a.out* file; both of these costs are modest.

### Prior Art

Postloading has been used on and off for over a decade, under a variety of names, but has rarely been published. Dennis Ritchie wrote one of the first, if not the first, postloaders for the Interdata 8/32, which did optimization similar to those described in the introduction. There were other such optimizers written, e.g. for the MC 68000 [RITC89]. Several commercial postloaders have been written, usually to help customers upgrade from older to newer hardware or to provide emulation of one kind of hardware on another. These programs are typically proprietary and have not been reported in the technical literature.

One postload optimizer described in the literature was done by David Wall [WALL86]. Wall assigns certain global variables to registers, and then rewrites modules to eliminate loads and stores of these variables.

Ritchie's and Wall's postloaders both have access to full relocation information, so strictly speaking they do not operate on a traditional *a.out* file. The size penalty of preserving this relocation information can be significant (from 15% to over 50% on some systems). In our postloading scheme, information to support postloading is included in the *a.out* file as a matter of course, and typically costs only a couple of percent in space. This allows optimization and other transformations to be done days or weeks after the loading.

### Technology and Implementation

The postloader technology consists of:

1. Modifications to the loader to put additional information into the *a.out* file.
2. A set of C programs that read and write *a.out* files, and support the addition, deletion, and modification of instructions.
3. For each specific postloader, a control program. These programs range from a few hundred to a few thousand lines, depending on the application.

There is one hard technical problem in postloading--mapping the old (pre-transformed) addresses to new ones. In order to do this transformation, it is necessary to:

- A. Find all all instructions and data in the original program that refer to text region addresses.
- B. Track these target addresses through the changes in the program text. Also, track any instructions that refer to the program text.
- C. After the transformations are completed, rewrite the instructions and data that refer to the text region to reflect the new addresses.

There are engineering issues in each of these steps. Step A is surprisingly hard. In a traditional UNIX environment, several things can conspire to make this problem undecidable. For example, many systems allow or encourage putting data in the text region (lazy hacker's read-only data). In the *a.out* file, these data words can look like instructions, and these instructions can appear to refer to text addresses; it is important to prevent the postloader from 'relocating' these data words. In the data region, the problem is reversed; it is necessary to identify pointers to functions

and other pointers to the text region (e.g., *switch* statement address tables) so they can be relocated after the transformations. Any general (e.g., non-heuristic) postloader must be able to make these identifications.

At Stardent, we dealt with this identification problem through a combination of restricting our options and adding information to the *a.out* file; the tradeoffs and strategies for other machines might well be different.

Note that the *ld* program knows about references to text addresses in the data region. We extended the *a.out* format to include a new region of postloading information. Whenever *ld* makes a relocation reference to the text region when relocating a data word, an entry is made in the new postloading region that captures the address of this relocation. The current postload region data is totally unoptimized; the region is just a list of addresses. Clearly, if *a.out* space were a problem we could have found frequently used structures (e.g., for *switch* blocks) and represented them more succinctly.

Because we committed to support postloading early in our development cycle, we simply banned data in program space. This allows us to pick up the majority of instruction text references by a simple scan of the text region looking for branches and calls. Had we allowed data in the text region, we still could have used the same strategy, but the postload information would have had to identify data blocks.

There is one class of instruction text references that cannot be found by a simple scan; these are 'load address' operations where a text address is being computed (e.g., a function pointer is generated to be passed into a subroutine). The problem is that the value generated may ambiguously represent either a data value (e.g., a bit mask) or a text address. Once more, *ld* knows the difference, and we again use the postload region to record 'load address' instructions that compute text addresses. This is harder in a RISC machine than in earlier machines, since 'load address' instructions are typically two instructions, and optimizations may separate these two instructions quite a bit in the program. We have, in a couple of cases, rejected low-level optimizations because they would have increased the complexity of postloading out of proportion to their benefit in code quality.

After the target addresses and their references are identified, the particular postloader transformations are applied. Here again, we could have allowed a general set of editing commands (cut, paste, block delete, etc.). Instead, we opted for a simpler scheme that was easy to understand and allowed postloading to be performed quickly. We make a single pass through the text region, and call a function, *transform* repeatedly with successive blocks of instructions. The first instruction of this block is typically the target of a branch. No other instruction in the block is a branch target, but the block may contain branches out of the block. The *transform* program is required to consider each instruction in turn; after possibly inserting new instructions before it, the instruction must either be explicitly deleted or explicitly copied. It is assumed that inserting instructions at the head of a block, before copying or deleting the first instruction, puts the new instructions after the label at the head of the block; to put instructions before a label, they must be inserted after the last instruction of the previous block.

By limiting the transformations in this way, it is possible to keep rather simple data structures for instructions and targets, and make simple changes in them as the transformations are done. Because the entire text region can be examined at any time, some very sophisticated transformations can be done with this rather simple mechanism. We have not felt much pressure to extend it.



The postloader technology is implemented by a set of routines that give a simple and fast interface that supports reading and writing of *a.out* files. A single call opens an *a.out* file and reads control information into a structure in memory, including the symbol table and the section headers. Individual sections are read by additional calls that read the entire section into memory and set up pointers and lengths in the main data structure. Postloading is supported by automatic computation of the text targets and the data structures that support the transformations. Finally, an extensive set of macros allows instructions to be picked up and 'cracked'; for example, macros tell whether an instruction is a branch, a floating point instruction, a load, a store, *etc.* Other macros extract the source and destination registers if needed.

When the transformation is complete, a single call will update the addresses and write a new *a.out* file. This file, in particular, contains all the information needed to postload it again if desired.

There are several subtle issues that arise in the UNIX framework. Many *a.out* files contain debugging information, and frequently this debugging information contains text addresses. In some cases, the text addresses are pretty subtly hidden (e.g., there are sometimes some wonderful optimizations on line number tables). In order to debug the postloaded program, this information must be transformed as part of the postloading process. At Stardent, we decided early on to abandon the COFF debugger support and we use our own format; not surprisingly, we made it easy in this format to identify and transform text addresses. It is likely that with many systems debugging support would be the most distasteful to program, difficult to debug, and error prone part of the postloader.

A traditional UNIX *strip* (1) command would cripple postloading. We initially did not offer *strip*. Soon, we were forced into provide a dummy version. Eventually, we did a 'real' one that simply maps all addresses in the symbol table to a single name: 'stripped'. This gives most of the security benefits of *strip* and some of the space benefits, but still allows much postloading to be done successfully.

## Applications

We have been delighted at the many and diverse uses we have found for postloading, which go far beyond optimization.

1. **Profiling.** The Stardent profiler is a postloader that adds counting and timing code to an *a.out* file, without need for recompilation or relinking. No special profiling libraries need be built and maintained. The symbol table is used to identify function entry points, and the profiler inserts code to call one of several timers (wall clock, UNIX user time, or a hardware tick counter). Optionally, timing counts for individual loop nests can be obtained if the program was compiled with an appropriate option. The profiler writes a *mon.out* file, like standard UNIX profilers; another postloader reads either the original or the transformed *a.out* and produces profiling statistics. The profiler also notices whether the loaded program is capable of multiprocessor operation, and adjusts its counting algorithm appropriately to collect statistics for each thread of the computation.
2. **Operation Counts.** A modification of the profiler counts floating point operations and collects dynamic statistics about average vector length and stride, percentage of operations that vectorize, etc.
3. **Performance Statistics.** We have written a number of postloaders to collect specific information related to architectural simulation, such as stack depth, cache and TLB behavior, etc. This is nice because we don't have to recompile and reload code to get these statistics.
4. **Flaky Hardware.** We have occasionally had to debug software with pre-production samples of chips that had significant errors. For example, in one situation we found that the last



instruction on a page could not be a load or a store. Using the postloader, we quickly added a NOP in this case. I don't know of another technology that would have allowed us to make progress in this situation. The big benefit here is that the main software (compilers, etc.) can continue to target the production architecture; the transient difficulties are handled by a postloader which can be quickly dumped when the hardware is corrected.

5. Flaky Software. RISC machines have some features (e.g., load and branch delays) that are difficult to handle in all situations where they arise. Moreover, the Stardent machines are multiprocessors, and synchronization instructions must be carefully managed to preserve correctness and yet allow the full performance promise of multiprocessing to be realized. We have written several postloaders that detect and/or correct common problems in development versions of software; this allows significant programs to be run and tuned while the compiler and libraries are being repaired, and also greatly aids in isolating compiler and library errors.
6. New Hardware Simulation. When moving to new hardware, we build a postloader that rewrites new code so it will run on current systems. This ensures that the new code and compiler are well debugged before entering the ring with the new hardware. Here again, our major software target is, and remains, the new hardware, while the postloader adjusts to reality.
7. New Hardware Transition. When shipping new hardware to our customers, a postloader can be used to make old programs run better without recompilation. For example, a postloader shipped with Stardent's Titan III product replaces calls to the square root subroutine in Titan II programs with the new hardware square root instruction. In going from Titan II to Titan III we have been able to add significant additional hardware functionality and streamline a lot of old features; in particular, we have changed the calling sequence conventions. Nevertheless, existing Titan II programs can still be run on Titan III after postloading, making the transition to new hardware easy for our customers.
8. Optimization. Postloaders are pretty fast, and have a lot of information available to them. We have experimented with a number of optimizers (including the one mentioned in the introduction), and will probably make one available in a future product offering. Note that there are some interesting optimizations possible with a postloader that are impossible prior to the loading process. For example, we can find functions that are called from only one place and streamline the calling sequence in this case. We can find functions that are never called and eliminate them, making the program smaller and improving locality of reference. This latter feature looks very interesting for languages such as C++ and Ada that tend to have packages with numerous small functions, many of which are not used in any particular application.
9. Miscellaneous. It is easy to write a new postloader (it frequently takes under an hour for a simple one), so a variety of 'one shot' postloaders have been written. One checked for a very obscure memory overwriting problem on every function call and return, and nailed the culprit the first time it was run. Another turned all floating point opcodes into illegal instructions, to find out where a particular program (that should have used integer only) was using floating point.

## Summary

We continue to think of new things to do with postloaders; it is very refreshing to have the *a.out* file no longer be a 'black box', but rather something that we can pick up and fondle. In fact, we have found postloading to be a partial antidote to the lack of source code.

Postloading works best when it is incorporated as part of the standard *a.out* format. The payoff for doing this is a much smoother transition to new hardware, and a simpler, more general implementation of such tools as profilers. The costs are modest, both in time and space.

## Acknowledgements

John Reiser, Bill Worley, Peter Eichenberger, John Wilkenson, and Mike McNamara were instrumental in inspiring, using, debugging, and encouraging this work.

## References

[CHOW86]

F. Chow, M. Himmelstein, E. Killian, L. Weber, "Engineering a RISC Compiler System", Proc IEEE Comcon, March 1986, San Francisco, 132-137.

[RITC89]

D. M. Ritchie, personal communication. None of the Bell Labs work on postloading has ever been published.

[SZYM78]

T. G. Szymanski, "Assembling code for machines with span-dependent instructions", CACM 21.4 (April 1978).

[WALL86]

D. Wall, "Global Register Allocation at Link Time", SIGPLAN '86 Compiler Construction Conference Proceedings, pp. 264-275.



Stephen C. Johnson  
*Stardent*

Stephen C. Johnson has a PhD. in Mathematics, but has spent his whole career in computing. He has worked in computer music, psychometrics, computer algebra, and VLSI design, but is probably best known for his contributions to UNIX, including *yacc*, *lint*, *at*, *spell*, and *pcc*. He and Dennis Ritchie did the first UNIX port. He has been a manager at AT&T Summit and Bell Labs, but left AT&T after nearly 20 years to join Stardent Computer, where he has held a variety of positions including VP Software. He has also been a member of the Usenix Board of Directors for nearly six years, and Treasurer for the last four. He is running for President of Usenix in 1990 and respectfully solicits your vote!

# Multiple Site Source Reconciliation

*Dodi Francisco  
Lois C. Price*

*TRW Financial Systems  
2001 Center Street  
Berkeley, California 94704*

## ABSTRACT

This paper describes the generic methodology developed at TRW Financial Systems to control the integration of source modifications made at multiple sites into a single master source which is the basis for a new release. The Multiple Site Source Reconciliation methodology has provided confidence that changes made between releases are not lost. The methodology minimizes the manual effort required by configuration management engineers and development engineers. Multiple Site Source Reconciliation has successfully supported several large projects.

## 1. Introduction

Multiple Site Source Reconciliation is a significant component of the sophisticated version control, release, and installation methodologies used at TRW Financial Systems. It has been successful in supporting three of our largest contracted projects over the past 20 months. Project software consists of application, UNIX, kernel, and configuration management tools. The three projects share some system-level characteristics, but have substantially different applications. The application source code for one of these projects is composed of over 5000 files containing over a million lines of code. The goal of configuration management is to have under source control all software resident on a production machine.

### 1.1. Requirements

The need for a reconciliation procedure arose from the requirement to support distinct software development environments at multiple sites. To keep the sites from diverging permanently, source modifications made at all sites must be incorporated into a single integrated master source. Periodically, the master source is generated, installed, and released to the sites, synchronizing all production and source environments.

The unique installation constraints and requirements are:

- A fast, if not immediate, solution to problems is required in the production environment. For one project, the production environment is a 24-hour, 7-day a week operation allowing no window for downtime.

- Production throughput and volume are such that situations which arise at the customer site often cannot be recreated at the master site. Therefore, problem resolution must occur at the customer site.
- Customer sites must have minimum dependence on the master site's source environment. If phone lines or machines are down at the master site (for instance, during backup or after a natural disaster), the customer sites must remain functional and supportable.
- Because of aggressive project scheduling at TRW Financial Systems and intense site operations, the sites and new development must be supported concurrently and continuously.

At each customer site, the separate software development environment was implemented to permit local quick fixes to the current release. Concurrently, development of new features takes place at the master site. Multiple Site Source Reconciliation ensures that subsequent releases include all fixes, enhancements and new features, regardless of the site where the modification was originally made.

Multiple Site Source Reconciliation requires only a small window of time during which the master site and a customer site are linked for file transfers. If this link is unavailable, the file transfers can be made by tape. During all other reconciliation activities, the sites operate independently.

## 1.2. Features

The Multiple Site Source Reconciliation methodology provides the following features:

- For each file under source control, its complete revision history from all sites is stored in a single master logfile.
- Any customer site source environment can be recreated at the master site.
- Configuration management and developer responsibility are well-defined and distinct.

## 2. Software Development Environment

The version control system at TRW Financial Systems is a substantially enhanced RCS system.

RCS provides the storage and retrieval mechanisms for incremental revisions of source files. Revisions of a file are stored in an associated logfile. Logfiles may also contain symbolic labels which enable the association of text strings with revision numbers. RCS supports the storage of multiple development paths, known as branches, within one logfile, and provides the capability to merge revisions that are on different development branches. The most recent revision on a branch is referred to as the leaf revision.

The Multiple Site Source Reconciliation implementation uses the RCS symbolic label facility and the multiple branch merge capability.

At TRW Financial Systems, software under development is divided into source control

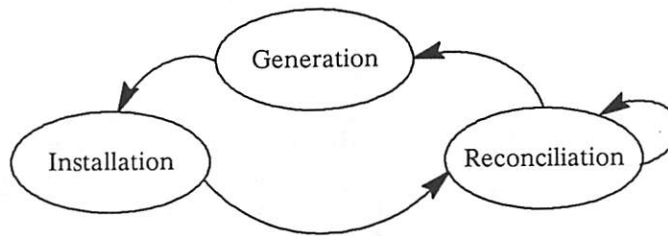
projects. Each source control project consists of a body of software that is developed, generated, and installed as a unit.

Each developer has a local source control environment in which to do development. A developer gets exclusive access to a revision via the check-out command, **cho**. After modifications have been tested, the developer checks in the new revision by executing the **chi** command. These commands are extensions of RCS which provide such features as a transaction log and a transparent update mechanism.

### 3. The Release Cycle

The baseline generation process formally generates software for test and release purposes. Baseline generation creates a version of the software from the most recent revisions stored in the source control project's logfiles. In each logfile, a symbolic label is defined to be the revision number used in the baseline version. The baseline installation process is an automated method to configure a production machine with baseline executables.

The release cycle consists of three steps: generation, installation, and reconciliation. The release cycle is depicted in Figure 1.



This shows the release cycle.

Figure 1

#### 3.1. Release Generation

A release version is a generated baseline that is installed at the customer sites. The defined symbolic label, referred to as the release label, is a component of the reconciliation process. At the time of release generation, a snapshot of the logfiles is created. A copy of the release version with the logfiles comprise the software which is taken to each site for installation.

#### 3.2. Installation

At each site, the installation process consists of distributing the release software to all production hosts and establishing the independent site software development environment from the release logfiles.



Source changes that have occurred since the last reconciliation are temporarily saved. These changes are manually incorporated by a developer into the new source control environment after configuration management has established it.

### 3.3. Multiple Site Source Reconciliation

Reconciliation is the method to incorporate into the master source environment all changes made at a site after a release installation. Reconciliation is an iterative process. Periodically, all changes that have been made at a site are transferred to the master site for processing. Typically, reconciliation occurs on a weekly basis until the time draws near for the next scheduled release. At that point, reconciliation occurs more frequently.

Since reconciliation occurs on a per-site basis, there may be several concurrent reconciliations.

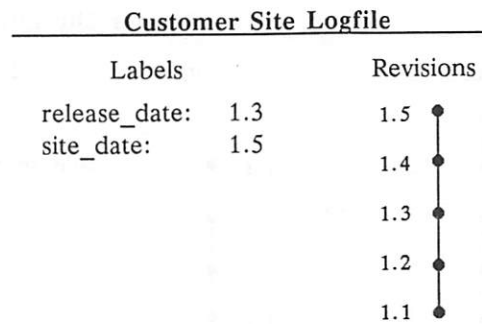
## 4. The Reconciliation Process

The following table shows the steps that comprise the reconciliation process, and indicates the responsible party (developer or configuration management) and the sites involved.

Reconciliation Step	Responsible Party	Required Sites
1. Site Revision Identification	CM	Customer
2. Site Logfile Transfer	CM	Customer & Master
3. Revision Audit Trail Creation	CM	Master
4. Developer Analysis	Developer	Master
5. Verification	CM	Master

### 4.1. Site Revision Identification

The `mark_reconcile` program is executed at the site. It defines a symbolic label in each logfile to indicate the last revision to be processed for this reconciliation pass. The format of a marker label is `site_date` and its value is the last checked-in revision. All revisions that have occurred since the last marker label, up to and including the new marker label, will be processed by this reconciliation pass. The release label serves as the initial marker label. If a file is not changed at the site, then all of the marker labels have the same value as the release label. Figure 2 shows a sample customer site logfile after it has been processed by `mark_reconcile`.



This diagram shows a customer site logfile after `mark_reconcile` has been executed one time. There have been two new revisions, 1.4 and 1.5 since the file was last released.

Figure 2

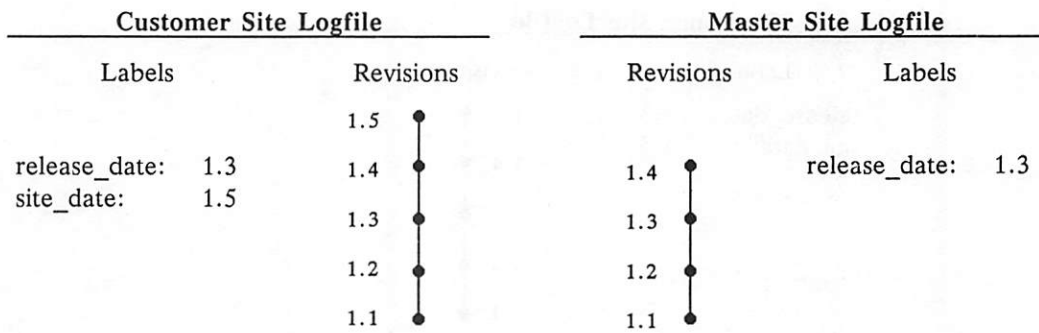
#### 4.2. Site Logfile Transfer

The customer site logfiles are transferred to the master site where further reconciliation processing will occur. The logfiles which contain new revisions are identified by querying a transaction log. This log is updated each time a new revision is checked-in to the site's source control environment. It is also possible to transfer all logfiles and determine which files include new revisions during the next step, revision audit trail creation.

The UNIX `tar` and `uucp` utilities perform the file transfer. All of the logfiles are combined into a tar file which is uucp'd to the master site, and then untar'd. This is the only step that requires the master and customer site to communicate. A tar'd tape can be used if the uucp link is unavailable.

#### 4.3. Revision Audit Trail Creation

The `graft_branch` program creates a revision audit trail in the master logfiles from the transferred site logfiles. The audit trail consists of the revisions which have been checked-in at the site since the release was installed. Figure 3 shows a sample of concurrent development at the customer and master sites as reflected in their respective logfiles.



This diagram shows a sample logfile as it exists at the customer and master sites when concurrent source changes occur. Revision 1.3 was the release revision. Two new revisions have been checked in at the customer site, 1.4 and 1.5. One change has been made at the master site, revision 1.4.

Figure 3

In each master logfile, there is a branch associated with each customer site. The site branch is an offshoot from the master branch that was generated for release. This offshoot occurs at the release revision.

Three symbolic labels defined in the master logfile control the processing of the site branch. For each site, these labels are:

*site\_branch*

The branch number.

*site\_leaf*

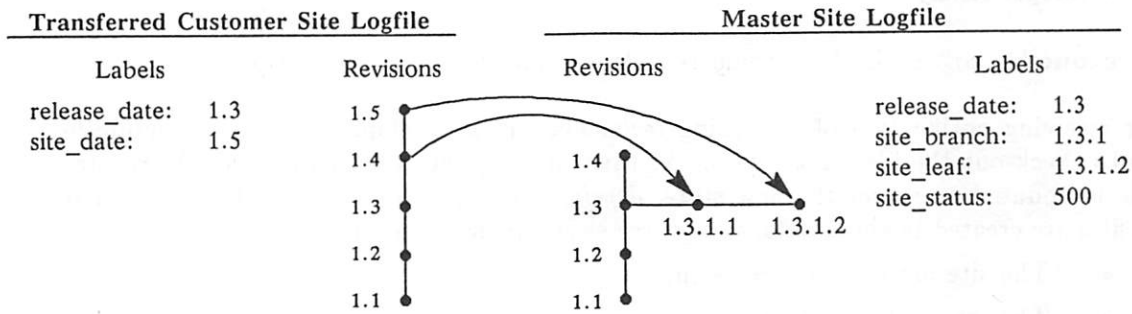
The last revision on the site branch.

*site\_status*

The site's reconcile status. Possible values are:

Status	Value
Reconciliation Pending	500
Developer Analysis In-Progress	501
Reconciliation Complete	599

Each customer site revision checked-in since the last reconciliation pass is checked-in to the master logfile as a new revision on the customer site branch. This is illustrated in Figure 4. By processing the marker labels, it is determined which revisions are to be grafted. To aid in the review of the customer site modification history, the check-in comment on each new revision indicates the original revision number.



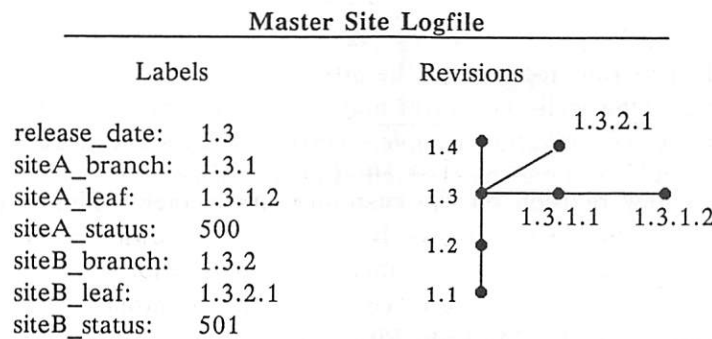
This diagram shows a master site logfile after the revision audit trail has been created. Customer site revisions 1.4 and 1.5 have been checked in as master site revisions 1.3.1.1 and 1.3.1.2, respectively.

Figure 4

The **graft\_branch** program also modifies the site's status label to *reconciliation pending*. This indicates that there are site revisions that need to be incorporated into the master branch.

**graft\_branch** automatically generates and sends a mail message to each developer that indicates the list of files that are pending reconciliation for which the developer is responsible. For each file, only the developer who checked-in the last revision at the site is notified.

When there are concurrent reconciles from several customer sites, each site is independently reconciled with the master branch. Figure 5 shows an example of a master site logfile with two customer site reconciliations.



This diagram shows a master site logfile with two concurrent reconciles, one from site A and one from site B.

Figure 5

#### 4.4. Developer Analysis

The **reconcile** program is the developers' sole interface to the reconciliation process.

After receiving notification of a pending reconciliation, a developer executes a **reconcile -out** to check-out the file for analysis. At this time, the site's status label in the master logfile is updated to reflect the new state, *developer analysis in-progress*. Three different workfiles are created in the developer's source environment as an aid:

- The site branch's leaf revision.
- The master branch's leaf revision.
- The join of the master leaf revision and the site leaf revision with respect to the release revision.

The RCS join option creates a file that consists of all changes made on the site branch and the master branch since the release revision. Overlapping changes are displayed in the same format employed by **diff**.

The developer determines the appropriate revision or modified revision for the master branch, and executes a **reconcile -in** to check-in the revision. At this time, the site's status label in the master logfile is set to *reconciliation complete*.

Note that the **-out** and **-in** options of **reconcile** parallel the programs used for normal development, **cho** (check-out) and **chi** (check-in).

The developer's analysis is critical: only the responsible developer knows what action is appropriate in the context of his/her work. Perhaps the site modifications have already been incorporated into the master source in a more elegant manner. Possibly, the site solution is the more desirable. Or, maybe the modification was a temporary change that does not need to be incorporated into the final product. Regardless, it is the developer who must make this decision.

#### 4.5. Verification

The **verify\_reconcile** program reports on the site status of one or more reconciliation passes. The program automatically generates mail for developers who are responsible for files that do not have a *reconciliation complete* status. Before the next release can be generated, the status of all files must be *reconciliation complete*.

### 5. Results

The results of incorporating the Multiple Site Source Reconciliation into the TRW Financial Systems release cycle are reflected in the following profile of one of our larger customers.

#### 5.1. Master Site Environment

The master site consists of ISI 68000 based machines. Most of the machines are running



4.3 BSD UNIX. Two of these machines serve as source control machines, providing storage for all source control data. The remaining machines support over forty developers.

## 5.2. Customer Site Environment

There are two customer sites which operate on a 24-hour, 7-day a week schedule. Each of these sites has up to thirty-five production computers and one source control computer. Each source control computer supports both configuration management and developer functions. Over ten developers support and maintain the site software.

## 5.3. Reconciliation Frequency

The following table illustrates the frequency of software releases and source reconciliations of the application code at one site over a 20 month period. These figures are nearly identical for the other site since our efforts have kept the sites synchronized. The other source control projects were also released and reconciled several times, though not as frequently as the application source.

	No. of Releases	No. of Reconciles	Avg No. of Files Processed
Jan-Jun	5	13	40
Jul-Dec	0	25	20
Jan-Aug	2	5	10

## 6. Conclusions

Over the 20 month period in which Multiple Site Source Reconciliation has been used, it has performed well under demanding conditions. It has been well accepted by project managers and developers. In the 7+ releases, there has not been a failure to merge a change into the master source. Once, a modification was not included in a release due to a developer's error in analysis. The problem was easily rectified by using the revision audit trail in the logfile.

We have been pleased to see that our generic methodology and tools have worked successfully in different project environments. Initially, reconciliation was used to verify that developers had incorporated changes in both the site and master source control environments. Developers were to manually enter any quick fix modifications at a customer site into the source environment at the master site. As confidence has grown in the system, the number of redundant quick fix modifications to the master source has decreased dramatically. Most developers wait until they are notified of a pending reconcile to make changes at the master site.

Planned enhancements to the **reconcile** program include making the creation of the join revision optional. Another option will enable the developer to select the customer site

leaf revision in a single step.

We are considering an extension which will help reduce lost changes even further. This program will analyze executables to detect any quick fix source changes that have not been checked into the version control system.

## 7. References

Walter F. Tichy, *An Introduction to the Revision Control System*, Programmer Supplement Documents PS1:13 of BSD4.3 Manuals



**Dodi Francisco**  
TRW

As Manager of the Configuration Management group at TRW Financial Systems, Dodi Francisco directed the design, implementation, and application of systems that control software development, generation, and installation. At Softyme, Inc., Ms. Francisco designed and developed a new product to automate customer service and technical support. She was responsible for the testing and trouble shooting of an electronic software distribution system and provided training and technical support for field installations of this system. Ms. Francisco has also worked with small businesses and non-profit organizations as an independent consultant for automated support systems, design, and development of training courses, technical writing, and database design and implementation.



**Lois C. Price**  
TRW

Lois C. Price defined the initial software configuration management methodology at TRW Financial Systems. She is currently involved in designing and implementing a major extension of the configuration management software. At MDS Qantel Corporation, she designed modifications to the UNIX kernel that provide memory protection through a ring architecture. Ms. Price also developed a file service package that unified and extended file system capabilities for the Qantel UNIX product. As a software project leader at Osborne Computer Corporation, Ms. Price was responsible for coordinating operating systems, diagnostics, and evaluation efforts and defining project methodology.

## CVS II: Parallelizing Software Development

*Brian Berliner*

*Prisma, Inc.  
5465 Mark Dabbling Blvd.  
Colorado Springs, CO 80918  
berliner@prisma.com*

### ABSTRACT

The program described in this paper fills a need in the UNIX community for a freely available tool to manage software revision and release control in a multi-developer, multi-directory, multi-group environment. This tool also addresses the increasing need for tracking third-party vendor source distributions while trying to maintain local modifications to earlier releases.

### 1. Background

In large software development projects, it is usually necessary for more than one software developer to be modifying (usually different) modules of the code at the same time. Some of these code modifications are done in an experimental sense, at least until the code functions correctly, and some testing of the entire program is usually necessary. Then, the modifications are returned to a master source repository so that others in the project can enjoy the new bug-fix or functionality. In order to manage such a project, some sort of revision control system is necessary.

Specifically, UNIX<sup>1</sup> kernel development is an excellent example of the problems that an adequate revision control system must address. The SunOS<sup>2</sup> kernel is composed of over a thousand files spread across a hierarchy of dozens of directories.<sup>3</sup> Pieces of the kernel must be edited by many software developers within an organization. While undesirable in theory, it is not uncommon to have two or more people making modifications to the same file within the kernel sources in order to facilitate a desired change. Existing revision control systems like RCS [Tichy] or SCCS [Bell] serialize file modifications by allowing only one developer to have a writable copy of a particular file at any one point in time. That developer is said to have "locked" the file for his exclusive use, and no other developer is allowed to check out a writable copy of the file until the locking developer has finished impeding others' productivity. Development pressures of productivity and deadlines often force organizations to require that multiple developers be able to simultaneously edit copies of the same revision controlled file.

The necessity for multiple developers to modify the same file concurrently questions the value of serialization-based policies in traditional revision control. This paper discusses the approach that Prisma took in adapting a standard revision control system, RCS, along with an existing public-domain collection of shell scripts that sits atop RCS and provides the basic

---

<sup>1</sup> UNIX is a registered trademark of AT&T.

<sup>2</sup> SunOS is a trademark of Sun Microsystems, Inc.

<sup>3</sup> Yes, the SunOS 4.0 kernel is composed of over a *thousand* files!

conflict-resolution algorithms. The resulting program, *cvs*, addresses not only the issue of conflict-resolution in a multi-developer open-editing environment, but also the issues of software release control and vendor source support and integration.

## 2. The CVS Program

*cvs* (Concurrent Versions System) is a front end to the RCS revision control system which extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories each containing revision controlled files. Directories and files in the *cvs* system can be combined together in many ways to form a software release. *cvs* provides the functions necessary to manage these software releases and to control the concurrent editing of source files among multiple software developers.

The six major features of *cvs* are listed below, and will be described in more detail in the following sections:

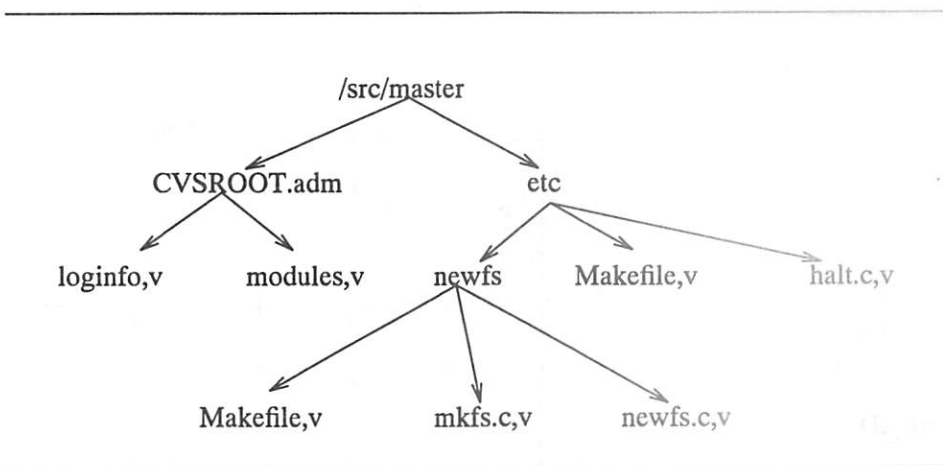
1. Concurrent access and conflict-resolution algorithms to guarantee that source changes are not "lost."
2. Support for tracking third-party vendor source distributions while maintaining the local modifications made to those sources.
3. A flexible module database that provides a symbolic mapping of names to components of a larger software distribution. This symbolic mapping provides for location independence within the software release and, for example, allows one to check out a copy of the "diff" program without ever knowing that the sources to "diff" actually reside in the "bin/diff" directory.
4. Configurable logging support allows all "committed" source file changes to be logged using an arbitrary program to save the log messages in a file, notesfile, or news database.
5. A software release can be symbolically tagged and checked out at any time based on that tag. An exact copy of a previous software release can be checked out at any time, *regardless* of whether files or directories have been added/removed from the "current" software release. As well, a "date" can be used to check out the *exact* version of the software release as of the specified date.
6. A "patch" format file [Wall] can be produced between two software releases, even if the releases span multiple directories.

The sources maintained by *cvs* are kept within a single directory hierarchy known as the "source repository." This "source repository" holds the actual RCS ".v" files directly, as well as a special per-repository directory (CVSROOT.adm) which contains a small number of administrative files that describe the repository and how it can be accessed. See Figure 1 for a picture of the *cvs* tree.

### 2.1. Software Conflict Resolution<sup>4</sup>

*cvs* allows several software developers to edit personal copies of a revision controlled file concurrently. The revision number of each checked out file is maintained independently for each user, and *cvs* forces the checked out file to be current with the "head" revision before it can be "committed" as a permanent change. A checked out file is brought up-to-date with the "head" revision using the "update" command of *cvs*. This command compares the "head" revision number with that of the user's file and performs an RCS merge operation if they are not the same.

<sup>4</sup> The basic conflict-resolution algorithms used in the *cvs* program find their roots in the original work done by Dick Grune at Vrije Universiteit in Amsterdam and posted to *comp.sources.unix* in the volume 6 release sometime in 1986. This original version of *cvs* was a collection of shell scripts that combined to form a front end to the RCS programs.



**Figure 1.**  
cvs Source Repository

The result of the merge is a file that contains the user's modifications and those modifications that were "committed" after the user checked out his version of the file (as well as a backup copy of the user's original file). cvs points out any conflicts during the merge. It is the user's responsibility to resolve these conflicts and to "commit" his/her changes when ready.

Although the cvs conflict-resolution algorithm was defined in 1986, it is remarkably similar to the "Copy-Modify-Merge" scenario included with NSE<sup>5</sup> and described in [Honda] and [Courington]. The following explanation from [Honda] also applies to cvs:

Simply stated, a developer copies an object without locking it, modifies the copy, and then merges the modified copy with the original. This paradigm allows developers to work in isolation from one another since changes are made to copies of objects. Because locks are not used, development is not serialized and can proceed in parallel. Developers, however, must merge objects after the changes have been made. In particular, a developer must resolve conflicts when the same object has been modified by someone else.

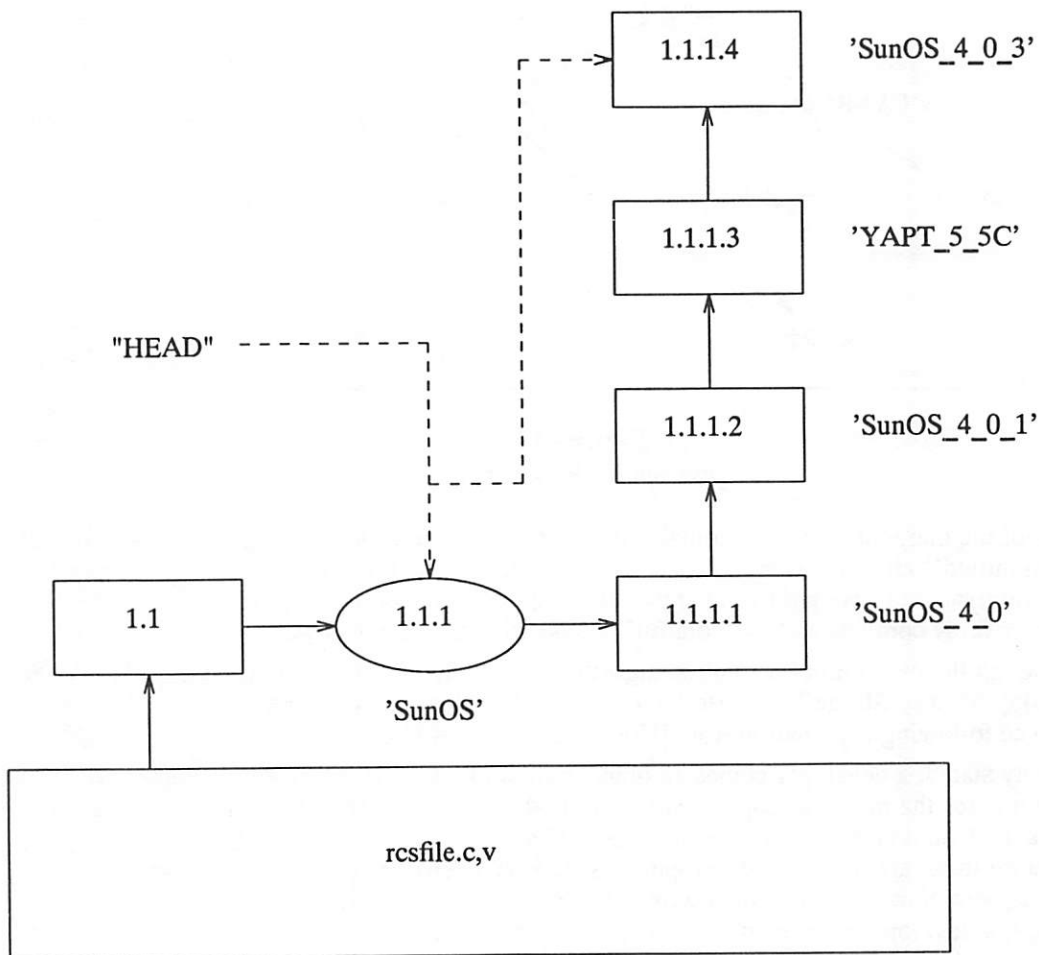
In practice, Prisma has found that conflicts that occur when the same object has been modified by someone else are quite rare. When they do happen, the changes made by the other developer are usually easily resolved. This practical use has shown that the "Copy-Modify-Merge" paradigm is a correct and useful one.

## 2.2. Tracking Third-Party Source Distributions

Currently, a large amount of software is based on source distributions from a third-party distributor. It is often the case that local modifications are to be made to this distribution, and that the vendor's future releases should be tracked. Rolling your local modifications forward into the new vendor release is a time-consuming task, but cvs can ease this burden somewhat. The **checkin** program of cvs initially sets up a source repository by integrating the source modules directly from the vendor's release, preserving the directory hierarchy of the vendor's distribution. The branch support of RCS is used to build this vendor release as a branch of the main RCS trunk. Figure 2 shows how the "head" tracks a sample vendor branch when no local modifications have been made to the file. Once this is done, developers can check out files and make local changes to the vendor's source distribution. These local changes form a new branch to the tree which is

<sup>5</sup> NSE is the Network Software Environment, a product of Sun Microsystems, Inc.



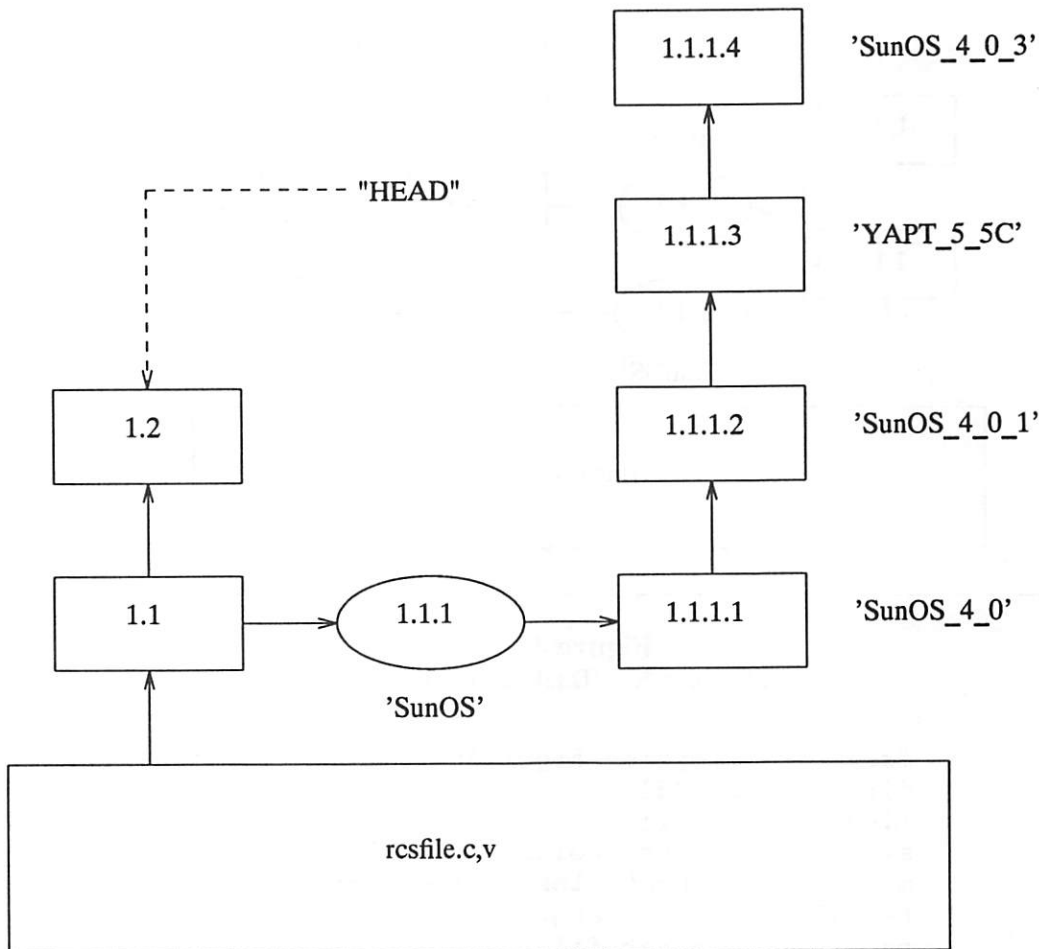


**Figure 2.**  
cvs Vendor Branch Example

then used as the source for future check outs. Figure 3 shows how the “head” moves to the main RCS trunk when a local modification is made.

When a new version of the vendor’s source distribution arrives, the **checkin** program adds the new and changed vendor’s files to the already existing source repository. For files that have not been changed locally, the new file from the vendor becomes the current “head” revision. For files that have been modified locally, **checkin** warns that the file must be merged with the new vendor release. The cvs “join” command is a useful tool that aids this process by performing the necessary RCS merge, as is done above when performing an “update.”

There is also limited support for “dual” derivations for source files. See Figure 4 for a sample dual-derived file. This example tracks the SunOS distribution but includes major changes from Berkeley. These BSD files are saved directly in the RCS file off a new branch.

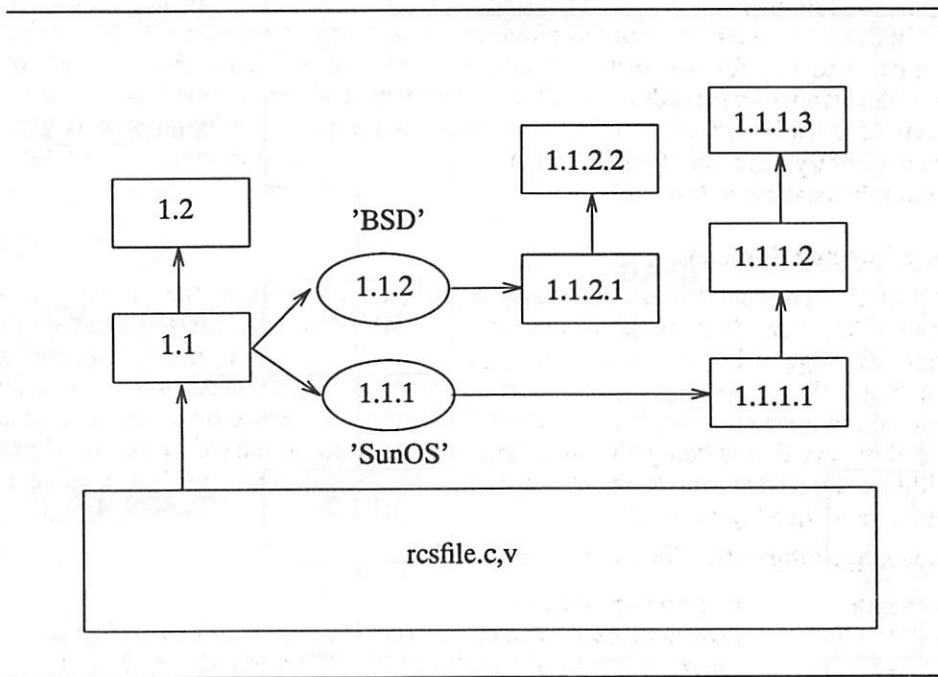


**Figure 3.**  
cvs Local Modification to Vendor Branch

### 2.3. Location Independent Module Database

cvs contains support for a simple, yet powerful, “module” database. For reasons of efficiency, this database is stored in **ndbm**(3) format. The module database is used to apply names to collections of directories and files as a matter of convenience for checking out pieces of a large software distribution. The database records the physical location of the sources as a form of information hiding, allowing one to check out whole directory hierarchies or individual files without regard for their actual location within the global source distribution.

Consider the following small sample of a module database, which must be tailored manually to each specific source repository environment:



**Figure 4.**  
cvs Support For "Dual" Derivations

```
#key      [-option argument] directory [files...]
diff      bin/diff
libc      lib/libc
sys        -o sys/tools/make_links sys
modules    -i mkmodules CVSROOT.adm modules
kernel     -a sys lang/adb
ps         bin Makefile ps.c
```

The "diff" and "libc" modules refer to whole directory hierarchies that are extracted on check out. The "sys" module extracts the "sys" hierarchy, and runs the "make\_links" program at the end of the check out process (the -o option specifies a program to run on checkout). The "modules" module allows one to edit the module database file and runs the "mkmodules" program on checkin to regenerate the **ndbm** database that **cvs** uses. The "kernel" module is an alias (as the -a option specifies) which causes the remaining arguments after the -a to be interpreted exactly as if they had been specified on the command line. This is useful for objects that require shared pieces of code from far away places to be compiled (as is the case with the kernel debugger, **kadb**, which shares code with the standard **adb** debugger). The "ps" module shows that the source for "ps" lives in the "bin" directory, but only *Makefile* and *ps.c* are required to build the object.

The module database at Prisma is now populated for the entire UNIX distribution and thereby allows us to issue the following convenient commands to check out components of the UNIX distribution without regard for their actual location within the master source repository:

```
example% cvs checkout diff
example% cvs checkout libc ps
example% cd diff; make
```

In building the module database file, it is quite possible to have name conflicts within a global software distribution. For example, SunOS provides two `cat` programs: one for the standard environment, `/bin/cat`, and one for the System V environment, `/usr/5bin/cat`. We resolved this conflict by naming the standard `cat` module “`cat`”, and the System V `cat` module “`5cat`”. Similar name modifications must be applied to other conflicting names, as might be found between a utility program and a library function, though Prisma chose not to include individual library functions within the module database at this time.

## 2.4. Configurable Logging Support

The `cvs` “`commit`” command is used to make a permanent change to the master source repository (where the RCS “`,v`” files live). Whenever a “`commit`” is done, the log message for the change is carefully logged by an arbitrary program (in a file, notesfile, news database, or mail). For example, a collection of these updates can be used to produce release notices. `cvs` can be configured to send log updates through one or more filter programs, based on a regular expression match on the directory that is being changed. This allows multiple related or unrelated projects to exist within a single `cvs` source repository tree, with each different project sending its “`commit`” reports to a unique log device.

A sample logging configuration file might look as follows:

```
#regex      filter-program
DEFAULT    /usr/local/bin/nfpipe -t %s utils.updates
^diag      /usr/local/bin/nfpipe -t %s diag.updates
^local     /usr/local/bin/nfpipe -t %s local.updates
^perf      /usr/local/bin/nfpipe -t %s perf.updates
^sys       /usr/local/bin/nfpipe -t %s kernel.updates
```

This sample allows the diagnostics and performance groups to share the same source repository with the kernel and utilities groups. Changes that they make are sent directly to their own notesfile [Essick] through the “`nfpipe`” program. A sufficiently simple title is substituted for the “`%s`” argument before the filter program is executed. This logging configuration file is tailored manually to each specific source repository environment.

## 2.5. Tagged Releases and Dates

Any release can be given a symbolic tag name that is stored directly in the RCS files. This tag can be used at any time to get an exact copy of any previous release. With equal ease, one can also extract an exact copy of the source files as of any arbitrary date in the past as well. Thus, all that’s required to tag the current kernel, and to tag the kernel as of the Fourth of July is:

```
example% cvs tag TEST_KERNEL kernel
example% cvs tag -D 'July 4' PATRIOTIC_KERNEL kernel
```

The following command would retrieve an exact copy of the test kernel at some later date:

```
example% cvs checkout -fp -rTEST_KERNEL kernel
```

The `-f` option causes only files that match the specified tag to be extracted, while the `-p` option automatically prunes empty directories. Consequently, directories added to the kernel after the test kernel was tagged are not included in the newly extracted copy of the test kernel.

The `cvs` date support has exactly the same interface as that provided with RCS, however `cvs` must process the “`,v`” files directly due to the special handling required by the vendor branch support. The standard RCS date handling only processes one branch (or the main trunk) when checking out based on a date specification. `cvs` must instead process the current “`head`” branch and, if a match is not found, proceed to look for a match on the vendor branch. This, combined with reasons of performance, is why `cvs` processes revision (symbolic and numeric) and date specifications directly from the “`,v`” files.

## 2.6. Building "patch" Source Distributions

cvcs can produce a "patch" format [Wall] output file which can be used to bring a previously released software distribution current with the newest release. This patch file supports an entire directory hierarchy within a single patch, as well as being able to add whole new files to the previous release. One can combine symbolic revisions and dates together to display changes in a very generic way:

```
example% cvcs patch -D 'December 1, 1988' \
                  -D 'January 1, 1989' sys
```

This example displays the kernel changes made in the month of December, 1988. To release a patch file, for example, to take the cvcs distribution from version 1.0 to version 1.4 might be done as follows:

```
example% cvcs patch -rCVS_1_0 -rCVS_1_4 cvcs
```

## 3. CVS Experience

### 3.1. Statistics

A quick summary of the scale that cvcs is addressing today can be found in Table 1.

Revision Control Statistics at Prisma as of 11/11/89	
How Many...	Total
Files	17243
Directories	1005
Lines of code	3927255
Removed files	131
Software developers	14
Software groups	6
Megabytes of source	128

**Table 1.**  
cvcs Statistics

Table 2 shows the history of files changed or added and the number of source lines affected by the change at Prisma. Only changes made to the kernel sources are included. The large number of source file changes made in September are the result of merging the SunOS 4.0.3 sources into the kernel. This merge process is described in section 3.3.

### 3.2. Performance

The performance of cvcs is currently quite reasonable. Little effort has been expended on tuning cvcs, although performance related decisions were made during the cvcs design. For example, cvcs parses the RCS "v" files directly instead of running an RCS process. This includes following branches as well as integrating with the vendor source branches and the main trunk when checking out files based on a date.

Checking out the entire kernel source tree (1223 files/59 directories) currently takes 16 wall clock minutes on a Sun-4/280. However, bringing the tree up-to-date with the current kernel sources, once it has been checked out, takes only 1.5 wall clock minutes. Updating the *complete* 128 MByte source tree under cvcs control (17243 files/1005 directories) takes roughly 28 wall clock minutes and utilizes one-third of the machine. For now this is entirely acceptable; improvements on these numbers will possibly be made in the future.



Prisma Kernel Source File Changes By Month, 1988-1989				
Month	# Changed Files	# Lines Changed	# Added Files	# Lines Added
Dec	87	3619	68	9266
Jan	39	4324	0	0
Feb	73	1578	5	3550
Mar	99	5301	18	11461
Apr	112	7333	11	5759
May	138	5371	17	13986
Jun	65	2261	27	12875
Jul	34	2000	1	58
Aug	65	6378	8	4724
Sep	266	23410	113	39965
Oct	22	621	1	155
Total	1000	62196	269	101799

**Table 2.**  
cvs Usage History for the Kernel

### 3.3. The SunOS 4.0.3 Merge

The true test of the `cvs` vendor branch support came with the arrival of the SunOS 4.0.3 source upgrade tape. As described above, the `checkin` program was used to install the new sources and the resulting output file listed the files that had been locally modified, needing to be merged manually. For the kernel, there were 94 files in conflict. The `cvs` “join” command was used on each of the 94 conflicting files, and the remaining conflicts were resolved.

The “join” command performs an `rcsmerge` operation. This in turn uses `/usr/lib/diff3` to produce a three-way diff file. As it happens, the `diff3` program has a hard-coded limit of 200 source-file changes maximum. This proved to be too small for a few of the kernel files that needed merging by hand, due to the large number of local changes that Prisma had made. The `diff3` problem was solved by increasing the hard-coded limit by an order of magnitude.

The SunOS 4.0.3 kernel source upgrade distribution contained 346 files, 233 of which were modifications to previously released files, and 113 of which were newly added files. `checkin` added the 113 new files to the source repository without intervention. Of the 233 modified files, 139 dropped in cleanly by `checkin`, since Prisma had not made any local changes to them, and 94 required manual merging due to local modifications. The 233 modified files consisted of 20,766 lines of differences. It took one developer two days to manually merge the 94 files using the “join” command and resolving conflicts manually. An additional day was required for kernel debugging. The entire process of merging over 20,000 lines of differences was completed in less than a week. This one time-savings alone was justification enough for the `cvs` development effort; we expect to gain even more when tracking future SunOS releases.

### 4. Future Enhancements and Current Bugs

Since `cvs` was designed to be incomplete, for reasons of design simplicity, there are naturally a good number of enhancements that can be made to make it more useful. As well, some nuisances exist in the current implementation.

- `cvs` does not currently “remember” who has a checked out a copy of a module. As a result, it is impossible to know who might be working on the same module that you are. A simple-minded database that is updated nightly would likely suffice.
- Signal processing, keyboard interrupt handling in particular, is currently somewhat weak. This is due to the heavy use of the `system(3)` library function to execute RCS

programs like `co` and `ci`. It sometimes takes multiple interrupts to make `cvs` quit. This can be fixed by using a home-grown `system()` replacement.

- Security of the source repository is currently not dealt with directly. The usual UNIX approach of user-group-other security permissions through the file system is utilized, but nothing else. `cvs` could likely be a set-group-id executable that checks a protected database to verify user access permissions for particular objects before allowing any operations to affect those objects.
- With every checked-out directory, `cvs` maintains some administrative files that record the current revision numbers of the checked-out files as well as the location of the respective source repository. `cvs` does not recover nicely at all if these administrative files are removed.
- The source code for `cvs` has been tested extensively on Sun-3 and Sun-4 systems, all running SunOS 4.0 or later versions of the operating system. Since the code has not yet been compiled under other platforms, the overall portability of the code is still questionable.
- As witnessed in the previous section, the `cvs` method for tracking third party vendor source distributions can work quite nicely. However, if the vendor changes the directory structure or the file names within the source distribution, `cvs` has no way of matching the old release with the new one. It is currently unclear as to how to solve this, though it is certain to happen in practice.

## 5. Availability

The `cvs` program sources can be found in a recent posting to the `comp.sources.unix` newsgroup. It is also currently available via anonymous ftp from "prisma.com". Copying rights for `cvs` will be covered by the GNU General Public License.

## 6. Summary

Prisma has used `cvs` since December, 1988. It has evolved to meet our specific needs of revision and release control. We will make our code freely available so that others can benefit from our work, and can enhance `cvs` to meet broader needs yet.

Many of the other software release and revision control systems, like the one described in [Glew], appear to use a collection of tools that are geared toward specific environments — one set of tools for the kernel, one set for "generic" software, one set for utilities, and one set for kernel and utilities. Each of these tool sets apparently handle some specific aspect of the problem uniquely. `cvs` took a somewhat different approach. File sharing through symbolic or hard links is not addressed; instead, the disk space is simply burned since it is "cheap." Support for producing objects for multiple architectures is not addressed; instead, a parallel checked-out source tree must be used for each architecture, again wasting disk space to simplify complexity and ease of use — punting on this issue allowed *Makefiles* to remain unchanged, unlike the approach taken in [Mahler], thereby maintaining closer compatibility with the third-party vendor sources. `cvs` is essentially a source-file server, making no assumptions or special handling of the sources that it controls. To `cvs`:

A source is a source, of course, of course, unless of course the source is Mr. Ed.<sup>6</sup>

Sources are maintained, saved, and retrievable at any time based on symbolic or numeric revision or date in the past. It is entirely up to `cvs` wrapper programs to provide for release environments and such.

<sup>6</sup> `cvs`, of course, does not really discriminate against Mr. Ed.<sup>7</sup>

<sup>7</sup> Yet.

The major advantage of `cvs` over the many other similar systems that have already been designed is the simplicity of `cvs`. `cvs` contains only three programs that do all the work of release and revision control, and two manually-maintained administrative files for each source repository. Of course, the deciding factor of any tool is whether people use it, and if they even *like* to use it. At Prisma, `cvs` prevented members of the kernel group from killing each other.

## 7. Acknowledgements

Many thanks to Dick Grune at Vrije Universiteit in Amsterdam for his work on the original version of `cvs` and for making it available to the world. Thanks to Jeff Polk of Prisma for helping with the design of the module database, vendor branch support, and for writing the `checkin` shell script. Thanks also to the entire software group at Prisma for taking the time to review the paper and correct my grammar.

## 8. References

- [Bell] Bell Telephone Laboratories. "Source Code Control System User's Guide." *UNIX System III Programmer's Manual*, October 1981.
- [Courington] Courington, W. *The Network Software Environment*, Sun Technical Report FE197-0, Sun Microsystems Inc, February 1989.
- [Essick] Essick, Raymond B. and Robert Bruce Kolstad. *Notesfile Reference Manual*, Department of Computer Science Technical Report #1081, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1982, p. 26.
- [Glew] Glew, Andy. "Boxes, Links, and Parallel Trees: Elements of a Configuration Management System." *Workshop Proceedings of the Software Management Conference*, USENIX, New Orleans, April 1989.
- [Grune] Grune, Dick. Distributed the original shell script version of `cvs` in the `comp.sources.unix` volume 6 release in 1986.
- [Honda] Honda, Masahiro and Terrence Miller. "Software Management Using a CASE Environment." *Workshop Proceedings of the Software Management Conference*, USENIX, New Orleans, April 1989.
- [Mahler] Mahler, Alex and Andreas Lampen. "An Integrated Toolset for Engineering Software Configurations." *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, Boston, November 1988. Described is the `shape` toolkit posted to the `comp.sources.unix` newsgroup in the volume 19 release.
- [Tichy] Tichy, Walter F. "Design, Implementation, and Evaluation of a Revision Control System." *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, September 1982.
- [Wall] Wall, Larry. The `patch` program is an indispensable tool for applying a diff file to an original. Can be found on uunet.uu.net in `ftp/pub/patch.tar`.



**Brian Berliner**  
*Prisma*

Brian Berliner has been a member of the Operating Systems group at Prisma, Inc. (Colorado Springs, Colorado) for over one year. His responsibilities have included cvs, a kernel source-level debugger, and file system performance improvements. Prior to joining Prisma, Brian spent three years in the Operating Systems group at Convex Computer Corporation (Richardson, Texas). He was responsible for Network File System (NFS) development and enhancements, UNIX kernel support and debugging, and the production of a HYPERchannel driver for the NETEX protocols. Brian earned his B.S. degree in Computer Science at the University of Illinois, Urbana-Champaign.

## Ada and Binary Unix Standards

*Mitchell Gart  
Alsys Inc.  
67 South Bedford Street  
Burlington, MA 01803  
(617) 270-0030  
gartm@ajpo.sei.cmu.edu*

### Background

Currently the two ways to distribute a software application that runs on Unix are either to distribute it in source form, and assume that all recipients have a compiler and development system with which to install it, or to compile and link the software before sending it. Using the latter approach, the software vendor must build, test, and maintain a different version of the software for each different Unix system to which the software is targeted. Even if the software is, for example, a C program that is source-code compatible between all the targets, the different binary versions are necessary because there are no binary Unix standards.

Distribution of binary programs is one area in which MS/DOS has a big advantage over Unix. Part of the reason is that MS/DOS only runs on the Intel 80x86 processor family. But a Unix software vendor can't build one binary version of a program for, say, Motorola 680x0 Unix, because the program must be re-compiled and/or re-linked in order to be moved between, for example, Sun and Hewlett-Packard machines that both are based on the 68020 and both run versions of Unix that support all the system calls made by the application.

Recently there have been two proposals for binary Unix software standards. Open Software Foundation (OSF) is proposing Architecture-Neutral Distribution Format (ANDF) [1], and AT&T and Unix International (UI) are proposing Application Binary Interface (ABI) [2]. To their credit, both organizations have recognized the problem, and are hard at work on solutions. The two solutions are very different from one another.

In the remainder of this paper, ABI and ANDF are explained in detail, then some special needs of the Ada language with respect to binary standards are mentioned, and finally each approach to binary standardization is evaluated with respect to its appropriateness for Ada.



## ABI

The ABI approach is to define a "generic" binary Unix standard that is common between machines, as the main standard, and then to define one "Supplement" with machine-specific details for each common architecture, for example the Intel 80386. The "generic" standard defines all system calls and parameters, the binary values for all system constants, the tree structure of the file system, and so on. The "Supplement" for each architecture defines the specific trap instruction or call that is used to make each system call, the register contents and stack layout upon entry to system calls, the register contents and memory layout upon program startup, and the memory layout for text, data, bss, "sbrk" area, and stack. The behavior of a C compiler in such areas as subprogram linkage conventions, structure layout, and register usage is specified. The mapping of machine instructions to Unix signals, and the stack layout when a signal is delivered, are exactly defined.

The 88Open group [5] has apparently been very successful in establishing a strong ABI that will apply to all Motorola 88000-family Unix systems, because the chip is new and the standard was established early. On the 80386 the current industry situation is that four major vendors have incompatible versions of Unix. AT&T and Interactive are binary-compatible with each other, but are different from Santa Cruz Operation (SCO) Xenix, Sun 386i, and IBM AIX for PS/2. The newly-released SCO Unix V.3 is compatible with AT&T and Interactive. Sun has pledged to become compatible with the AT&T ABI when Sun's Unix V.4 is released sometime in 1990.

## ANDF

The ANDF request for technology calls for "A specification for providing architecture-neutral software distribution. Possible examples include specification of an intermediate compiler format, encrypted source, or tagged executable." The comments in the rest of this paper refer to the "intermediate compiler format".

The idea of the intermediate compiler format is that a compiler front end for a language such as C or Ada would translate the source program into intermediate code. An application program would be translated in this way and then distributed. On the Unix machine where the application is to run, an "installer" program would translate the intermediate code into machine instructions and link it with system libraries.

The front end would have no knowledge of the target machine, and the "installer" program would have no knowledge of the language in which the application was written. One front end per source language could be written, and one "installer" per target machine architecture could be written.

The theory is that compilers can be written in this way - with a front end that is entirely independent of the machine architecture and a back end that is independent of the source language. A universal intermediate language is supposed to be able to express all language semantics in machine- and language-independent terms. Once this intermediate system is established, support for a new high level language can be added by writing a new front end for that language, and the new front end will then work for

all the machines that are currently targeted. A new "installer" can be added for another machine architecture, and then the system will support all of the languages for which front ends exist.

Presumably the assumption is that the "installer" program can be small enough that it can reside on every machine running Unix, and can be inexpensive and easy to write for each new machine architecture.

Another universal translation system called UNCOL [3] was first written about in 1958. The basic ideas are remarkably similar to ANDF. Only the high level languages and target machines have changed.

### **ABI and ANDF contrasted**

The current situation is that each application must be re-built (re-compiled and/or re-linked) for each processor and each version of Unix, for example Sun Unix on 80386, AT&T Unix on 80386, IBM Unix on 80386, HP Unix on 68020, and so on.

The situation with a strong ABI standard would be that an application could be built once for all versions of 80386 Unix, then rebuilt for all versions of Unix on another machine architecture, and so on.

Finally, the situation with ANDF would be that an application would be built once, distributed to all target Unix systems, and then installed on each machine that has an "installer" program.

The first observation to be made is that the current situation, with no binary distribution standards, is chaotic. ABI is a "halfway" solution that would dramatically improve the current situation, but would still require different binary versions of an application for different machine architectures. ANDF is more of an "ultimate" solution, allowing software to be built once and then shipped to all possible customers, although it does require an "installer" program on each target machine.

Another observation that can be made is that the two approaches are not necessarily mutually exclusive. ANDF in fact almost requires an underlying ABI so that only one installer program can be written for each target machine architecture. It is thus possible to have ANDF and also ABI. A software vendor could either distribute a program in the ANDF intermediate format, leaving it to be installed on each machine, or could translate it into ANDF, then run the installer program, and finally distribute the results of the installation, one version per machine architecture.

### **What Ada needs from a binary Unix standard**

Ada works well on Unix. There are many different validated compilers from several vendors, for several machine architectures. For an explanation of how Ada is implemented on one Unix system see [6].

The Ada program execution model on Unix is different from C in a few ways that effect binary standardization. If Ada programs are to be portable without recompilation or relinking, Unix systems must be exactly alike in ways that are not usually important for C. The following areas are of concern:

**Exact signal context** - When an Ada program causes a machine fault, for example division by zero or null pointer dereference, Unix sends a signal to the program and the Ada runtime system gets control and "propagates" the exception to a handler in the user's program. The runtime must typically look at information that is pushed on the stack at the point the signal is received to find the program counter and other registers at the time of the exception. The runtime almost never directly returns to the place where the signal occurred, it almost always goes to another place in the Ada program. Often the runtime must change the machine registers and pop several frames from the stack.

This area is notoriously unportable, even between Unix systems on the same machine architecture. For Ada program portability, a Unix standard must specify the exact mapping from machine instructions to signals, on a per machine basis, and the exact layout of the information that is pushed when a signal handler gets control. This information is, of course, dependent on the machine architecture.

The Ada runtime system doesn't use "setjmp" and "longjmp" to establish exception handlers and go to them because of efficiency. It is common for an Ada program to declare exception handlers for different scopes within the application. It might be possible for the compiler to generate calls to "setjmp" upon entry and exit of each scope with an exception handler, but this would be very costly. With the scheme that is currently used, no extra code is executed to establish exception handlers, the only cost occurs if an exception is actually raised during program execution.

**Task stacks** - Ada allows an application program to define logically concurrent tasks. The most common implementation of Ada multi-tasking on Unix is to map an entire Ada program to one Unix process, with each Ada task having its own execution stack inside the memory of the Unix process. The most reasonable way to implement this is to use the main Unix stack for the main Ada task, and then to dynamically allocate additional Ada task stacks, as needed, with "malloc" or "sbrk".

In a version of Unix that includes separate threads of control within a process, such as Mach, it would be most reasonable to map each Ada task to a thread.

**Main stack extension** - On some Unix implementations the way to extend the main stack is not documented and not portable. This is a problem with different 68000-family Unix systems, but not with the 80386. Typically the local 68000 C compiler generates a certain instruction to create a stack frame. If the stack overflows, the Unix kernel recognizes this instruction and silently extends the stack. If any other instruction makes a reference beyond the end of the stack, a signal is sent to the program.

A binary standard must explicitly document the way the main Unix stack is extended, and it must be the same on all Unix systems in a given machine architecture.

**Code "rx" data "rwx"** - Ada obviously needs to be able to execute program code or text. It also needs to be able to read program code memory, for exception handling. A common implementation of exception handling includes having the Ada runtime system

locate the instruction that caused a signal, and then read that instruction, to get more information.

Ada also obviously needs to be able to read and write program data memory. In a few cases it also needs to be able to execute an instruction that is in data memory. One example is that in some cases the 80387 floating point coprocessor will cause an exception. The Ada runtime system will get control and as part of the recovery from the exception will build another floating point instruction in memory, and then execute that instruction. This is not allowed by the current 80386 ABI, see below.

**Finding a file's name** - The predefined Ada input/output packages define a function `NAME` which takes as parameter a file descriptor and returns a string. The name that is returned is supposed to be the file's full Unix pathname. It is possible but costly to implement `NAME` for files that are opened explicitly, but there is no standard way (other than searching the entire disk) to implement it when one of the standard Unix file descriptors is redirected. It would be nice if Unix helped by providing a system call which returned the name of a file descriptor, or alternatively if there were a way for an application program to access its full command line, including redirections.

#### Ada and the current 80386 ABI

Overall, the current 80386 ABI is an excellent basis for writing programs in Ada or another language that will be portable in binary form between different versions of 80386 Unix. The areas mentioned above and below are the few exceptions where the standard needs work to support Ada. It should be noted that in most cases the changes that are suggested are certainly not contradictory to Unix, and don't even really change Unix. Perhaps the best way to characterize them is to say that they address areas that Ada cares about and C doesn't care about.

**Exact signal context** - The standard says that in the future the signal context may change, and applications may not rely on the fact that it will stay the same. (The signal context is in "level 2" of the standard - "level 1" features are guaranteed not to change and "level 2" features may change in the future.) Furthermore, the standard says that in the future the signal context may be hidden from the application by the Unix kernel.

Ada really needs to know exactly where a program was executing when a signal occurred. It needs to know the state of the machine registers, and needs the ability to "return" to a different place in the program, having changed the registers. The standard should be defined in a way that this information is available and is unlikely to change between Unix releases.

**Task stacks** - Unfortunately the current version of the 80386 ABI specifically disallows a program to set its stack pointer into the "sbrk" memory area. Its model of program execution is that the "sbrk" area of memory and the "stack" area are separate, and the program's stack pointer must always be inside the Unix idea of the "stack" area.

In fact, on the current version of AT&T 80386 Unix, a program may execute with its stack pointer inside the "sbrk" area under most circumstances, but if a signal arrives the Unix kernel checks the stack pointer. If the stack pointer is in a zone the kernel thinks



is legal it delivers the signal, otherwise it kills the program. With this restriction it is possible to allocate several Ada task stacks within the Unix "stack" area, but it is clumsy. One way to organize the Ada program is to give every stack, including the main one, a fixed size. This sacrifices the automatic Unix main stack extension mechanism.

To better accommodate Ada (and probably other languages that allow multiple tasks or multiple threads of execution) the model should allow a program to execute with its stack pointer pointing into the Unix "stack" area or the Unix "sbrk" area.

**Code "rx" data "rwx"** - The current 80386 ABI disallows execution out of a program's data memory (although the current AT&T 80386 Unix system allows it). To accommodate Ada, the model should allow execution of data memory. This feature is also probably necessary for Lisp, since Lisp allows a program to build a list of instructions and then execute the list.

**C bias** - The 80386 Supplement is a mixture of requirements for an 80386 Unix implementation and requirements for an 80386 C compiler. It would be clearer if the two areas were separated into "architecture" and "C" sections. The current section on parameter passing only includes C parameter modes, not Ada "out" or Pascal "var" parameters. This is understandable, but it should be separated and labelled as a C-language specific section, so that it can later be extended to include other languages.

**Floating point emulation** - The standard should state whether the Unix kernel emulates the 80387 floating point coprocessor when no coprocessor is present, and it should state what floating point instructions and execution modes are supported.

## Ada and ANDF

The author has several reservations about the practicality of ANDF:

**Cost to change existing compilers** - The intermediate format between front end and code generator is a very fundamental part of the architecture of any compiler. For a compiler to completely change the intermediate format that is generated is tantamount to re-designing the compiler from the beginning. This is not cheap. The Alsys Ada compiler, for example, has taken more than 100 person-years of work to develop, and consists of over 300,000 lines of Ada source.

**Intermediate language too general** - It is probably possible to specify an intermediate language (IL) that would be usable for one source language such as C, with multiple machine targets:

```
-> 80386
C -> IL -> 68000
-> ...
```

or it is probably possible to specify an intermediate language that would be appropriate for several languages and just one machine architecture:

```
Ada    ->
```



```
C      -> IL -> 68000
Fortran ->
...
```

but the author doesn't think the two approaches can successfully be combined into one super intermediate language which would take into account the needs of all computer languages and also all machine architectures. UNCOL was not a success, and p-code [7] was a system that was widely used but never resulted in efficient code generation.

**Extensibility** - An intermediate language that was designed for C and later extended to include Ada would probably not be successful, for example because of Ada checks and exception handling semantics. These areas and others specific to the Ada language were fundamental parts of the design of intermediate languages for Ada (see next section) and could not have easily been added as an afterthought or an extension.

**Bigness of the installer program** - The major divisions of the Alsys Ada compiler are:

analyzer -> AIL -> expander -> CGIL -> code generator

The front end ("analyzer") produces one intermediate language, called AIL (Abstract Intermediate Language), which is basically an Ada tree plus symbol table. The expander translates the AIL plus symbol table into another intermediate language called CGIL (Code Generator Intermediate Language). The analyzer has a few target machine dependencies, and the expander has many dependencies. (The reason an Ada front end has machine dependencies is because the front end is responsible for emitting compile-time error messages, and some messages related to primitive data types depend on the target machine.)

If ANDF were used for Ada, probably it would be closer to the level of the AIL than the CGIL, with the ANDF installer corresponding to the Alsys expander plus code generator for a given target machine. This would at least put most of the target machine dependencies in the installer.

The Alsys Ada compiler is a little bigger than 2 megabytes of code. This bigness is common for Ada compilers. The expander is the biggest of the three major compiler sections. Putting the expander plus code generator into the ANDF installer makes the installer a very big program, which will have to be present on all machines where an application is to be distributed. Since the ANDF idea is that there is just one language-independent installer, this size applies to all ANDF installers.

**Conclusions about ANDF** - The author does not think ANDF is a good idea when applied to Ada software. Some reasons are specific to Ada and others apply to all languages. Generally, the author thinks that ANDF would not be as flexible as its proponents intend and it would not be so easy to add a new language or a new machine. ANDF would lead to the domination of a few languages on a few common machine architectures. The situation would be especially difficult for languages with unusual features not taken into account in the ANDF design, or for machines with unusual instruction sets.

ANDF could be more acceptable if it were proposed as a research project instead of as an industry standard to be implemented immediately. Standards should be based on proven technology, not on unproven ideas.

## POSIX

Unix International and Open Software Foundation have both said they will support the IEEE POSIX [4] standard. POSIX will be a subset of AT&T Unix System V.4, and also a subset of OSF/1. It would be beneficial for the advancement of Unix if OSF and UI could agree on a standard similar to ABI, which would provide binary compatibility between System V.4 and OSF/1 applications which only use the routines defined in POSIX.

Users and software developers should demand that OSF and UI work on a binary standard based on POSIX or some other subset that is common between the two organizations.

## Conclusions

The current situation for distributing binary Unix programs is chaotic. Serious work is finally being done to establish industry standards in this important area, but the two approaches being proposed are completely different.

The author doesn't think the ANDF "intermediate compiler format" is a workable idea in the immediate future.

An ABI which would allow binary applications to run on different vendors' Unix systems on the same machine architecture is a good, practical, workable idea. Applications would have to be recompiled when moving across machine architectures.

ABI needs small modifications to support Ada. These modifications are not contrary to Unix or C, but rather involve adding more details to the standard than would be necessary for C.

Users and software developers should demand that OSF and UI work on a binary standard based on POSIX or some other subset that is common between the two organizations.

## Acknowledgements

Thanks to the following people for reviewing this paper or otherwise providing input or help: John Barnes, Con Bradley, Pascal Cleve, Bob Eachus, Dave Emery, Mireille Gart, Bernard Henin, Larry Rowe, the Usenix program committee, Sri Vasudevan, and Bill Wulf.

## References

- [1] ANDF refers to Request for Technology for "Architecture-Neutral Distribution Format", Open Software Foundation, Cambridge, MA.
- [2] ABI refers to "System V Application Binary Interface", Second Industry Review Draft, AT&T, March, 1989. The Intel 80386 supplement is "Intel386 Architecture Supplement", Intel Corp, March 23, 1989. In some AT&T and Intel documents the names "System V BCS" and "386BCS" are used to refer to the same set of documents. BCS stands for Binary Compatibility Standard.
- [3] Strong, J. et. al., "The Problem of Programming Communication with Changing Machines: a Proposed Solution" Communications of the ACM, 1:8 and 1:9, August-September, 1958, pp 12-18 and 9-15. UNCOL stands for *Universal Computer Oriented Language*.
- [4] *POSIX: IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE, New York, 1988.
- [5] "88Open Binary Compatibility Standard", Release 1.0, 88Open, Wilsonville, Oregon, February 1989.
- [6] Gart, M., "Targeting Ada to 68000/Unix", Usenix Technical Conference, Denver, Colorado, January 1986.
- [7] Barron, D.W., *Pascal - The Language and Its Implementation*, John Wiley, Chinchester, England, 1981, pp. 125-170.



**Mitchell Gart**  
*Alsys Inc.*

Mitchell Gart received an M.S. degree in computer science from the University of California at Berkeley in 1979. He currently works on 80386 Unix Ada compilers for Alsys, Inc. Previously he has worked for Cii-Honeywell Bull and Ampex. He is co-author of *Software Portability*, McGraw-Hill Books, New York, 1989. He is Rationale Editor for the IEEE 1003.5 POSIX/Ada committee.



# **TAE Plus: Transportable Applications Environment Plus**

## *A User Interface Development and Management System*

Martha R. Szczur  
NASA/Goddard Space Flight Center  
Greenbelt, Maryland 20771  
mszczur@postman.gsfc.nasa.gov  
(301) 286-8609

Karl R. Wolf  
Century Computing Inc.  
1014 West Street  
Laurel, Maryland 20707  
kw@cen.com (301) 953-3330

### **ABSTRACT**

The Transportable Applications Environment Plus (TAE Plus™) is a portable user interface development and management system, which provides (1) an intuitive WYSIWYG WorkBench for prototyping and designing an application's user interface, integrated with (2) tools for efficiently implementing the designed user interface and (3) effective management of the user interface during an application's active domain. During the development of TAE Plus, many design and implementation decisions were based on the state-of-the-art within graphics workstations, windowing systems and object-oriented programming languages, and this paper shares some of the problems and issues experienced during implementation and porting among different Unix-based graphic workstations. The paper concludes with open issues and a description of the next development steps planned for TAE Plus.

### **TAE PLUS OVERVIEW**

TAE Plus is a tool for designing, building and tailoring an application's user interface (UI) and for controlling the designed UI throughout the application's execution. The main component of TAE Plus is a WYSIWYG user interface designer's "WorkBench" that allows an application developer to interactively construct the look and feel of an application screen by arranging and manipulating "interaction objects" (e.g., radio buttons, menus, icons, dials, strip charts, etc.).

Once the application's screen has been designed, the WorkBench saves the user interface details in a resource file. TAE Plus includes runtime services, Window Programming Tools (WPTs), which are used by application programs to display and control the user interfaces designed with the WorkBench. Since the WPTs access the resource file during execution, the user interface details remain independent from the application code, allowing changes to be easily made to the look and feel of an application without recompiling or relinking the software. To change the user interface, the designer returns to the WorkBench, dynamically makes the modifications, and the resource files are automatically updated. The next time the application is run, the modifications will be in effect. Figure 1



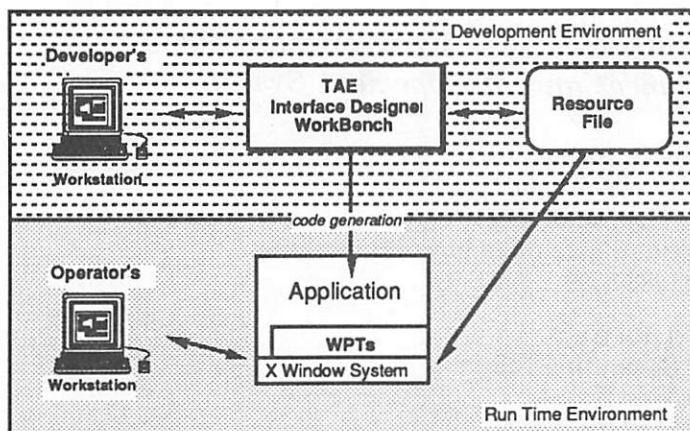


Figure 1. TAE Plus Structure

## INTERACTION OBJECTS AS BUILDING BLOCKS

The basic building blocks for developing an application's user interface are a set of interaction objects. All visually distinct elements of a display that are created and managed using TAE Plus are considered to be interaction objects. Within TAE Plus, interaction objects fall into three categories: user-entry objects, information objects and data-driven objects. User-entry objects are mechanisms by which an application can acquire information and directives from the end user. They include radio buttons, text entry fields, scrolling text lists, pulldown menus and push buttons. Information objects are used by an application to instruct or notify the user, such as contextual on-line help information displayed

in a scrollable static text object or brief status/error messages displayed in a "bother box". Data-driven objects are vector-drawn graphic objects which have been "connected" to an application data variable, and elements of their view change as the data values change. Examples are dials, thermometers, and strip charts. Figure 2 illustrates the current set of TAE Plus interaction objects (which are referred to as *items* in the WorkBench). For advanced screen designs, these items can be grouped or composed into larger interaction objects, called *panels* by the WorkBench.

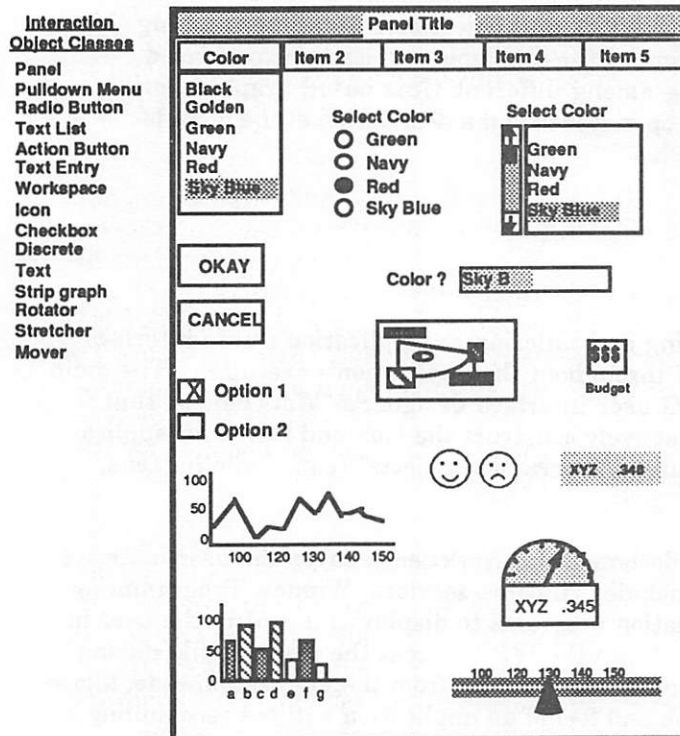


Figure 2. Current set of TAE interaction objects

The use of interaction objects offers the application designer/programmer a number of benefits with the expected payoff of an increase in programmer productivity: The interaction objects

provide a consistent look and feel for the application's user interface, which translates into reduced end-user training time, more attractive screens, and an application which is easier to use. Another key benefit is that since the interaction objects have been thoroughly tested and debugged, the programmer is able to spend more time testing the application, and less time verifying that the user interface behaves correctly. This is particularly important considering the complexity of some of the objects, and the programming effort it would take to code them from scratch.

## WORKBENCH SCENARIO

The WorkBench provides an intuitive environment for defining, testing, and communicating the look and feel of an application system. As a designer tool, it provides the following key features:

- Customization and direct manipulation of user interaction objects
- Application code generator
- Capability to dynamically define "connections" between interaction objects
- Rehearsal capability to "try out" sequencing of the user interface design
- Icon editor and support for raster objects
- Capability to dynamically draw and define data-driven graphic objects
- Undo capability
- Help icon/button on-line support

Let's walk through a simple design scenario to get a feel for how the WorkBench operates. The application is a hardware monitoring task for a satellite data handling facility and the designer is going to lay out the user interaction in which the operator is prompted for a hardware channel number. Once the operator selects a channel, a new panel appears with a realtime sliding bar object displaying the amount of data flowing through the channel. Figure 3 shows a rough hand-drawn sketch of the two panels which are to be designed with the WorkBench in this scenario.

Functionally, the WorkBench allows an application designer to dynamically lay out an application screen, defining its static and dynamic areas. The tool provides the designer with a choice of pre-designed interaction objects and allows for tailoring, combining and rearranging of the objects. To begin the session, the designer needs to create the base window into which interaction objects will be specified. He/she selects **New Panel** from the main WorkBench panel, which displays the panel specification panel (Refer to Figure 4)

where the designer specifies presentation information, such as the title, font, color, and optional on-line help for the panel *Monitor*.

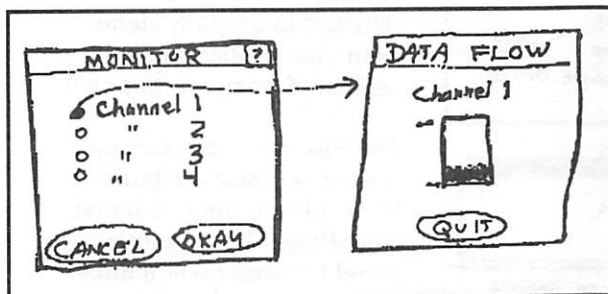


Figure 3. Hand-drawn sketch of application's user interface to be created with WorkBench

The designer is now ready to define the interaction items to reside in the panel. He/she selects **New Item** from the WorkBench main panel and is presented with the item specification window. The designer defines both the presentation information and the context information. The item

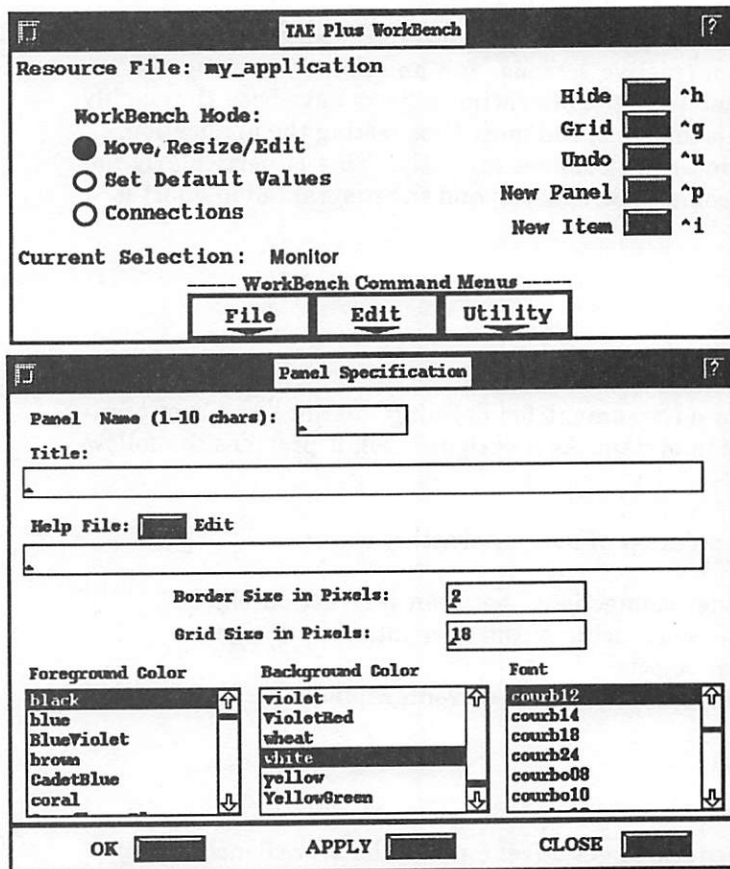


Figure 4. WorkBench's Main Panel and Panel Specification

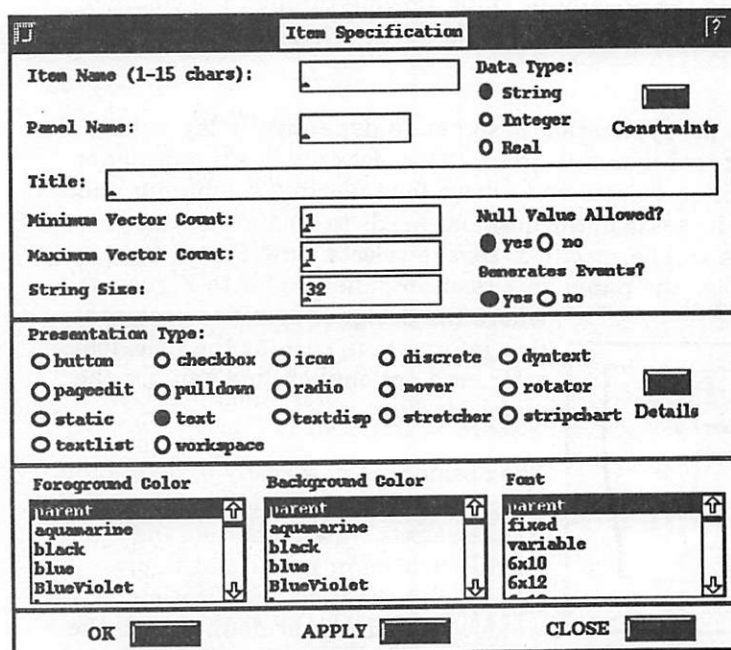


Figure 5. Defining the interaction objects to reside in a panel.

specification window has an associated *Constraints* (i.e., context) window within which the labels for each entry of a radio button bank object are specified (refer to Figure 5). For the scenario we are following here, the designer has created a radio button bank for the channel numbers, a *cancel* and *okay* button and a panel help icon. For icon support, the WorkBench has an Icon Editor, within which an icon can be drawn, edited and saved.

The designer also has the option of retrieving a "palette" of items (by selecting **File...Include** from the WorkBench panel). From this collection of previously created items, the designer can select and copy appropriate objects. The ability to reuse items saves programming time, facilitates trying out different combinations of items in the prototyping process, and contributes to standardization of the application's "look and feel". If an application system manager wanted to ensure consistency and uniformity across an entire application's UI, all developers could be instructed to use only items from the application's palette of common items.

The designer goes through the same process to build the realtime display panel, *DataFlow*. This simple panel is made up of a data-driven stretcher item, selected from a pre-defined palette of "output objects",

and a *quit* button. The WorkBench provides a drawing tool [3] within which the static background and dynamic foreground of a data-driven object can be drawn, edited and saved. Once the object is created, the designer identifies presentation attributes for the object (i.e. the color thresholds, maximum/minimum, delta).

Most often an application's UI will be made up of a number of related panels, sequenced in a meaningful fashion. Through the WorkBench, the designer defines the interface "connections". These links determine what happens when the user selects a button or a menu entry. The designer attaches "events" to interaction items and thereby designates

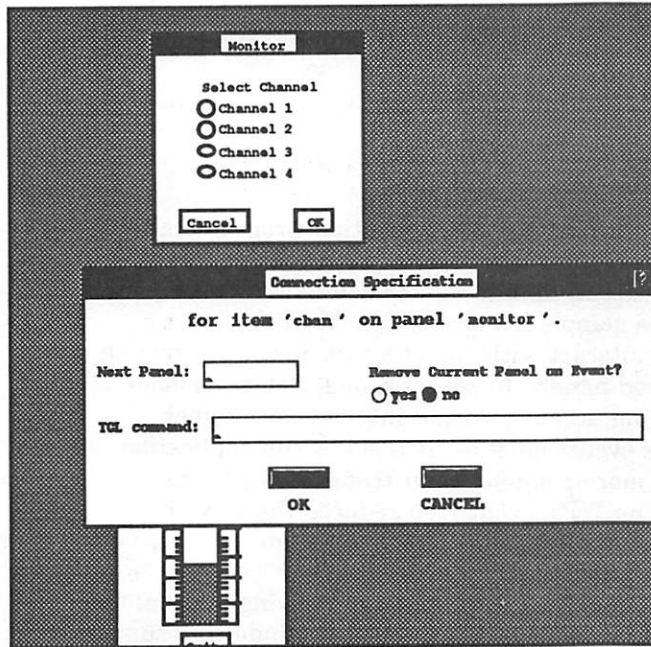


Figure 6.

Using the WorkBench to define "connections".

window in which the appropriate information can be entered. The designer can then define any button item or icon item to be "the" help item for the panel (in the scenario we are following, it would be the Help icon in the panel *Monitor*). During the application operation, when the end-user clicks on the question mark item, the cursor changes to a "?". The end-user then clicks on the panel itself or any item in the panel to bring up a help panel containing the associated help text.

Having designed the layout of panels and their attendant items and having threaded the panel and items according to their interaction scenario, the designer must then be able to preview (i.e., to rehearse) the interface's operation. With this potential to "test drive" an interface, to make changes, and to dry-run again, iterative design becomes part of the prototyping process. When the designer selects the rehearse option (by selecting **Utility....Rehearse** from the WorkBench panel), the screen is cleared and the WorkBench goes through the entire sequence as if the application were executing. With the rehearsal feature, the designer can evaluate and refine both the functionality and the aesthetics of a proposed interface. After the rehearsal, control is returned to wherever the designer left off in the WorkBench and he/she can either continue with the design process or save the defined UI in a resource file (by selecting **File....Save** from the WorkBench Menu).

what panel appears and what program executes when an event is triggered. Events are triggered by user-controlled I/O peripherals (e.g., point and click devices or keyboard input). In Figure 6, the designer has specified links causing the *Dataflow* panel to appear when the end user selects the option marked Channel 1 and the process *Flowcompute* to be executed. In turn, *Flowcompute* is the application process containing the data variable that drives the variations in appearance of the item *BarSlide*.

TAE Plus also offers an optional help feature which provides a consistent mechanism for supplying application-specific information about a panel and any interaction items within the panel. In a typical session, the designer elects to edit a help file after all the panel items have been designed. Clicking on the edit help option brings up a text editor



Developing software with sophisticated user interfaces is a complex process, mandating the support of varied talents, including human factors experts and application program specialists. Once the UI designer (who may have limited experience with actual code development) has finished the UI, he/she can turn the saved UI resource file over to an experienced programmer. As a further aid to the application programmer, the WorkBench's "generate" feature (Utility...Generate) produces a fully annotated and operational body of code which will display and manage the entire WorkBench-designed UI. Currently, source code generation of C, Ada and TCL are supported, with bindings for Fortran and C++ expected in later TAE Plus releases. The programmer can now add additional code to this template and make a fully functional application. Providing these code "stubs" helps in establishing uniform programming method and style across large applications or within a family of interrelated software applications.

## WINDOW PROGRAMMING TOOLS (WPTs)

The Window Programming Tools (WPTs) are a package of application program callable subroutines used to control an application's user interface. Using these routines, applications can define, display, receive information from, update and/or delete TAE Plus panels and interaction objects (refer to Figure 7 for a sample list of WPTs). WPTs support a modeless user interface, meaning a user can interact with one of a number of interaction objects within any one of a number of displayed panels. In contrast to sequential mode-oriented programming, modeless programming accepts, at any instance, a number of user inputs, or *events*. Because these multiple events must be handled by the application program, event-driven programming can be more complex than traditional programming. The WorkBench's auto-generation of the WPT event loop reduces the risk of programmer error within the UI portion of an application's implementation.

The WPT package utilizes the X Window System™ [6] as its base windowing system. One of the strengths of X is the concept of providing a low-level abstraction of windowing support (Xlib), which becomes the base standard, and a high-level abstraction (X toolkits), which has a set of interaction objects (called "widgets" in the X world) that define elements of a

UI's look and feel. The current version of TAE Plus (V4.1) is implemented with the X11.3 version using the X Toolkit. The initial approach is to base our default set of interaction objects on the HP widget set delivered with the generic M.I.T. delivery of X, while supporting an open architecture that allows adding to the widget set. A "cookbook" explaining the steps to be taken to replace/add widgets and update the WorkBench is included with the TAE Plus documentation set. Thus, an application development group can choose or create a specific widget set,

Wpt_AddEvent	Add other sources for input/output/exception
Wpt_BeginWait	Display busy indicator cursor
Wpt_CloseItems	Close Items on a Panel
Wpt_ConvertName	Get the X Id of a named window
Wpt_Endwait	Stop displaying busy indicator cursor
Wpt_Init	Initializes interface to X Window System
Wpt_ItemWindow	Gets the window Id of the window containing a parameter
Wpt_MissingVal	Indicates if any values are missing
Wpt_New Panel	Displays a user interface panel
Wpt_NextEvent	Gets next panel-related event
Wpt_PanelErase	Erases the displayed panel from the screen
Wpt_PanelMessage	Displays message in "Bother Box"
Wpt_PanelReset	Resets object values to initial values
Wpt_PanelTopWindow	Gets panel's parent shell window Id
Wpt_PanelWidgetId	Return the Widget Id of a Wpt Panel Widget
Wpt_PanelWindow	Returns the X Id of a panel
Wpt_ParmReject	Generates a rejection message for a given value
Wpt_ParmUpdate	Updates the displayed values of an object
Wpt_Pending	Check if a WptEvent is pending from X, Parm or file.
Wpt_RemoveEvent	Remove a previously registered event
Wpt_SetTimeOut	Set/Cancel timeout for gathering Wpt events.
Wpt_ViewUpdate	Updates the view of a parameter on a displayed panel

Figure 7. The Window Programming Tools (WPTs)



integrate it into TAE Plus and design user interfaces with their preferred style while using TAE Plus to provide consistency in design process and in program management. As different "widget sets" become popular and/or standard, TAE Plus will migrate towards supporting these widgets. For instance, in the first quarter of 1990, work on a version of TAE Plus which builds Motif™ [4] compliant user interfaces will begin.

The WPTs also provide a buffer between the application program and the X Window System services. For instance, to display a WorkBench-designed panel, an application makes a single call to `Wpt_NewPanel`. This single call translates into a function that consists of about 2800 lines of C code and makes about 50 calls to X Window System routines. For the majority of applications, the WPT services and objects supported by the WorkBench provide the necessary user interface tools and save the programmer from having to learn the complexities of programming directly with X. This can be a significant advantage, especially when considering that the full set of 26 Wpt routines consist of 37,000 lines of C++ code and make a total of between 300-400 X calls.

### PROTOTYPING IN TAE COMMAND LANGUAGE (TCL)

To provide an easy method for displaying and manipulating the newly designed user interface, we created a simple set of commands ("WPT" commands) within the TAE Command Language (TCL). TCL offers a high-level set of commands used to invoke and manage application functions. Commands can be invoked dynamically during an interactive session or used to build command procedures. An advantage TAE Plus has over some other UI development systems is that it does not just support the user interface component of an application, but has a full set of integrated tools to fully support an application, either a prototype or an operational version. These services include parameter manipulation, message logging, logon/logoff procedure, data file I/O, operating system services, scripting capability, session logging, procedure building capability, on-line help, and user-site tailoring of TAE Plus commands. Because the user interface tools are integrated with general purpose application management services, the application need not be tied tightly to a particular operating system or computer.

Since TCL is an interpreted language, the commands can be used to prototype an application without having to recompile or relink every time a change is made. Just as with WPT routines used by application programs, the WPT commands can be used to directly define panels and items, or they can be used to access WorkBench-generated resource files that contain pre-defined panels and items. While the intended use of these commands is for prototyping, if the overhead performance of executing TCL commands is acceptable, then command procedures using WPT commands would be appropriate for operational systems.

### IMPLEMENTATION

The TAE Plus architecture is based on a separation of the user interaction management from the application-specific software. The current implementation is a result of having gone through several prototyped and beta versions of a WorkBench and user interface support services during the 1986-89 period, as well as building on an existing application management system, the original TAE (affectionately referred to as "TAE Classic"). [5]

TAE Classic, which was designed in 1980, provides the application developer with a general purpose command language and a standard presentation of menus, parameter

prompting, error/information messages and help screens for the ASCII terminal. (Note: There are 220 active TAE Classic user sites.) To support the abstraction of the user interface from the application, the definition information is stored and manipulated in editable text files. The structure of these data files proved robust and flexible enough to support the definition of the TAE Plus interaction objects. The binary form of these data definition files are the resource files generated by the WorkBench and accessed by the WPTs during run-time.

TAE Plus still includes all the "Classic" ASCII terminal user interface support, as well as an option to display these standard menus, parameter entry and on-line help screens with a graphic "facelift". This enables the same application to operate on ASCII-based terminals and/or graphics workstations with a Classic and/or Facelift user interface, respectively. This facelift feature also aids in migration of existing TAE Classic applications into the world of graphics.

The "Classic" portion of the TAE Plus code ( $\approx 60,000$  LOC) is implemented in the C programming language. In selecting a language for the WorkBench and the WPT runtime services, we felt a "true" object-oriented language would provide us with the optimum environment for implementing the TAE Plus graphical user interface capabilities. (See Chapter 9 of Cox [1] for a discussion on the suitability of object-oriented languages for graphical user interfaces.) We selected C++ as our implementation language for several reasons [9]. For one, C++ is becoming increasingly popular within the object-oriented programming community. Another strong argument for using C++ is the growing availability of existing, public domain, X-based object class libraries. Utilizing an existing object library is not only a cost saver, but also serves as a learning tool, both for object-oriented programming and for C++. Delivered with the X Window System is the *InterViews* C++ class library and a drawing utility, *idraw*, both of which were developed at Stanford University. [2,3] The *InterViews* C++ class library has many attractive features. The class structure has gone through several major iterations and the current design is clean. The *idraw* utility is a sophisticated direct manipulation C++ application, which we integrated into the WorkBench to support creating, editing and saving the graphical data-driven interaction objects.

## PORTABILITY ISSUES AND UNIX

Throughout the design and development of TAE Plus, one of our primary goals has been to be "portable" over a wide range of hardware platforms. It was a requirement that TAE Plus operate on various UNIX systems and VAX/VMS. There are three primary software areas identified to be the most nonportable -- file manipulation, process control and interprocess communication. The software modules to support these areas are localized and tailored to the individual operating system of each host environment. With the proper use of tailored *include* files, TAE Plus ports between Unix machines with few problems. The TAE installation procedure asks whether you want a BSD or System V implementation. In strategic places in the source code, the `"#ifdef SYSV"` compiler directive is used to conditionally compile for either BSD or SYSTEM V systems. TAE Plus also runs on DEC VAX/VMS systems with much the same technique.

When porting among different hardware platforms, the host system's method of storing binary information is always of key concern. Since TAE Plus's resource files are binary, we provide a utility to produce a straight ASCII equivalent of the file. This file can then be transferred to any platform that accepts the ASCII character set and then converted back into a binary file, which can be read by TAE Plus applications (including the WorkBench) operating on the target platform.

As mentioned earlier in this paper, the C language was selected for implementing TAE Classic and it has proven to be an efficient and standard language across different hardware platforms. The C++ code has not been quite as portable as we would have liked. There are several differences, even syntactical, among the various C++ compilers. We therefore decided to initially limit our support to just two compilers. They are the GNU C++ (v1.35.1) and the OASYS Designer C++ compiler. There are some tradeoffs between the two compilers. The OASYS compiler is easy to install, and is a supported commercial product. The GNU C++ is somewhat challenging to install, but it is available at no cost from the Free Software Foundation, Inc. The OASYS compiler precompiles into C and the host C compiler is used to generate the object code. The GNU C++ compiler directly generates object code, so it has faster "code change to object file" turnaround time. Since the compilers generate mutually incompatible object code, only one compiler can be used when compiling the TAE Plus code and any C++ applications programs using the WPT runtime libraries. In the next release of TAE Plus, the Oregon C++ compiler will also be supported.

While the integration of existing public-domain software into the TAE Plus package is a distinct time and cost saver, it does create an added complication when it comes to porting to other machines. In the case of *InterViews* and *idraw*, which are written in C++, we become dependent on using the same C++ compilers as the developers of these adopted packages. When we want to extend our support of TAE Plus to include a different compiler, the testing and debugging process involves not only our own code, but includes the challenge of testing and debugging all of the adopted software. We anticipate that as C++ grows in popularity, the pressure for compiler standardization will eventually eliminate these current compiler issues.

There is another compiler-related issue, which becomes an interesting porting complication. The WorkBench generates source code, which displays and manages the designed user interface, in C, Ada, and TCL. While C and TCL port between machines with no problems, one of the prime areas that is not standardized in Ada compilers is function calls to foreign languages. Since the main activity of the Ada WPTs is to bundle the parameters into the required format needed to send them to the corresponding C++ WPT, the Ada code generation function within the WorkBench ends up being compiler dependent. In the current release of TAE Plus, the Ada WPT bindings only support the Verdex Ada compiler on Unix platforms.

The single most important factor contributing to the portability of TAE Plus is the X Window System. When we first began experimenting with developing a user interface that looked and behaved in a similar fashion on different graphic workstations, there were no industry standard window management systems. Prototype versions of TAE Plus attempted to use the host window management system, and we learned that the vendor window systems were based on such widely varying concepts that portability of our software was extremely limited ( $\approx$  fifty percent rewrite). Therefore, when the X Window System was introduced and emerged into an industry standard, the need to port window management functions was eliminated. Generally, if a graphic workstation supports the Xlib and X Toolkit and operates either Unix or VMS, TAE Plus can be ported to it with reasonable ease.

## AVAILABILITY and MAINTAINABILITY

The last two years have been spent in prototyping and developing beta versions of the TAE Plus. In January 1990, an "industrial strength" version of TAE Plus (Version 4.1) became available at a nominal license fee from COSMIC, the NASA distribution center



located at the University of Alabama. While TAE Plus base development and testing is done on a Sun workstation under Unix, within our R&D laboratory at GSFC, we also port and validate TAE Plus with formal acceptance testing on the following Unix workstations: Apollo, Vaxstation II, Decstation 3100, HP9000, and Macintosh II (A/UX). TAE Plus has also been ported by individual users to the Masscomp, IBM PS/2 under AIX, Sun4 Sparcstation, and Silicon Graphics Iris. (Note: Although this paper is directed to Unix-oriented readers, it should be mentioned that TAE Plus is also available on the Vaxstation II under VMS.) Beta versions of TAE Plus were distributed to over 325 world-wide sites in 1989.

Maintenance of a software system is a key factor in its success, and while every system is maintainable; *how easy it is to maintain* is the real issue. We knew when we began that TAE Plus was targeted for wide application utilization and for different machines, so ease of maintenance has always been important. By providing the application-callable WPTs and WPT function commands, applications are isolated from the windowing system. Thus, when the latest release or next generation windowing system shows up, only the WPTs will require updating or rewriting; the application code will not be affected.

User support is another facet of maintainability. Since the first release of TAE Classic in 1981, we have provided user support through a fully staffed Support Office. This service has been one of the primary reasons for the success of TAE. Through the Support Office, users receive answers to technical questions, report problems, and make suggestions for improvements. In turn, the Support Office keeps users up-to-date on new releases, provides training sessions, and sponsors user workshops and conferences. This exchange of information enables the Project Office to keep the TAE software and documentation "in working order" and, perhaps most importantly, take advantage of user feedback to help direct our future development.

## NEXT STEPS

The current TAE Plus provides a powerful and much needed tool for the continuum of software engineering -- from the initial design phases of a highly interactive prototype to the fully operational application package. However, there is still a long list of enhancements and new capabilities that we will be adding to TAE Plus in future releases. Features included on the "Wanted List" are extensions to the interaction objects, refinements and improvements to the realtime performance of the data-driven objects; integration with the Open Software Foundation's (OSF) Motif™; ports to new workstation platforms; on-line tutorial and training tools; introduction of hypermedia technology; integration of expert system technology to aid in making user interface design decisions; and implementation of additional user interface designer tools, such as a WYSIWYG graph builder.

## CONCLUSION

With the emergence of sophisticated graphic workstations and the subsequent demands for highly interactive systems, the user interface becomes more complex and includes multiple window displays, the use of color, graphical objects and icons, and various selection techniques. Prototyping of different user interface designs, thus, becomes an increasingly important method for stabilizing concepts and requirements for an application. At GSFC, we had the requirement to provide a tool for prototyping a visual representation of a user interface, as well as establish an integrated development environment that allows prototyped user interfaces to evolve into operational applications.

We feel TAE Plus is fulfilling this role by providing a usable, generalized, portable and maintainable package of development tools.

TAE Plus is an evolving system and its development will continue to be guided by user-defined requirements. To date, each phase of TAE Plus's evolution has taken into account advances in virtual operating systems, human factors research, command language design, standardization efforts and software portability. With TAE Plus's flexibility and functionality, we believe it can contribute to both more advances and more standardization in user interface development system technology.

## ACKNOWLEDGEMENTS

TAE Plus is a NASA software product being developed by the NASA/Goddard Space Flight Center and by Century Computing, Inc. The work is sponsored by the NASA Office of Space Science and Applications and the Office of Space Operations.

TAE is a registered trademark of National Aeronautics and Space Administration (NASA). It is distributed through NASA's distribution center, COSMIC, (404) 542-3265. For further information, contact the TAE Support Office at GSFC, (301) 286-6034.

## REFERENCES

1. Cox, Brad J., OBJECT ORIENTED PROGRAMMING, AN EVOLUTIONARY APPROACH, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
2. Linton, Mark, Calder, Paul R., "The Design and Implementation of InterViews", Proceedings of the C++ Workshop, USENIX, November, 1987, pp.256-273.
3. Linton, Mark A., Vlissides, John M., Calder, Paul R., "Composing User Interfaces with Interviews", IEEE COMPUTER, February, 1989.
4. Open Software Foundation, Inc., OSF/Motif™ Programmer's Reference Manual, Revision 1.0, 1989
5. Perkins, D.C., Howell, D.R., Szczur, M.R., "The Transportable Applications Executive -- an interactive design-to-production development system", DIGITAL IMAGE PROCESSING IN REMOTE SENSING, edited by J-P Muller, Taylor & Francis Publishers, London, 1988.
6. Scheifler, Robert W., Gettys, Jim., "The X Window System", MIT Laboratory for Computer Science, Cambridge, MA., October 1986.
7. Seidewitz, Ed., "Object-Oriented Programming in Smalltalk and Ada", Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Conference, October, 1987.
8. Stroustrup, Bjarne, THE C++ PROGRAMMING LANGUAGE, Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.
9. Szczur, Martha R., Miller, Philip, "Transportable Applications Environment (TAE)



Plus: Experiences in "Object"ively Modernizing a User Interface Environment", Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Conference, September 1988.

10. TAE Plus V4.1 Documentation Set, Century Computing, Inc., NASA Contractor Documentation, January 1990.

11. Wegner, Peter, "Dimensions of Object-Based Language Design", Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Conference, October, 1987.



**Martha R. Szczur**

Ms. Szczur has over twenty years experience in the design and development of interactive advance data information systems, user interface technology, graphics and image processing systems. For the past six years, Ms. Szczur has been the Project Manager of the Transportable Application Environment (TAE) and is involved in the planning and design of prototype data systems for several next-generation NASA space information systems.



**Karl R. Wolf**

Mr. Wolf was born in New Haven, Connecticut and received his BS in Information and Computer Science from the Georgia Institute of Technology (76). He also received an MS in Computer Science from the Johns Hopkins University (79). Throughout his career, Mr. Wolf has been involved in such varied specializations such as database administration, computer graphics, image processing and for the past few years in user interface management and TAE Plus.





## *The USENIX Association*

*T*he USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *login.*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

*Computing Systems*, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710

Telephone: 415 528-8649  
Email: [office@usenix.org](mailto:office@usenix.org)

